SEPTEMBER 1989: VOLUME 2, ISSUE 6

THE MAGAZINE FOR AMIGA PROGRAMMERS

Canada $4.25
USA $3.50

FOR THE Amiga Transactor

Creating your own '.device' ***by Steve Simpson***

High-speed serial communications ***by Mike Schwartz***

Colour cycling made easy ***by Hobie Orris***

PLUS:

- Writing a compiler - the parser *by Chris Gray*
- Seth: a Chaos game *by Vic Wagner*
- How random are your random numbers? *by Eric Giguère*
- The Icon language *by Scott Ballantyne*
- Regular columns by Jim Butterfield and Don Curtis

Reviews: *X-Specs 3D* stereoscopic glasses, *Ami...Alignment System* disk drive alignment software

Transactor by Anthony Kashinn

0 29074 12122 0 9

Coming To A Screen Near You.

If you're ready to become your own studio producer, there's only one desktop video system that will bring NTSC and S-VHS to your Amiga's screen. The Scanlock system by VidTech. Other systems give you a genlock that's "either-or." They either support NTSC or S-VHS. But not both.

Two For One

With Scanlock, you get two genlocks for the price of one. So you can migrate from NTSC to the newer S-VHS format without investing in another system. We also offer a Scanlock model that is compatible with the International PAL TV format, including its new S-VHS version.

Scanlock's dual-genlock system gives you broadcast-quality video at consumer prices. And our system offers all the bells and whistles, including front-panel fingertip controls, multiple monitor connections for viewing and editing, and independent fade controls for both reference and Amiga videos. If your Amiga is supporting a load of peripherals, we provide a connection that lets you use an optional external power supply.

Scanlock is a breeze to use. A "glitch-free" vertical interval switch lets you insert, reverse and remove reference video and Amiga graphics at the touch of a finger. And you can switch between NTSC, S-VHS and analog RGB formats without moving cable connectors. When you're done editing, you can resume normal Amiga operation with the flip of a switch.

Don't Get Locked Out

So if you're ready to become your own producer, choose the desktop video system that doesn't lock you into one genlock — the Scanlock system by VidTech. Coming to a screen near you.

Call or write us today: VidTech International, Inc., 2822 NW 79th Ave., Miami, Florida 33122. Telephone 800-727-2261 or 305-477-2228. Fax 305-591-1651. Prices:NTSC-$995, PAL-$1,095. Distributed by Southern Technologies, Dallas, Texas.

VidTech

InterComputing Inc.

2100 N. Hwy 360, Suite 2101
Grand Prairie, TX 75050-1015

In business since 1984; Amiga support since 1985
Member of Dallas Better Business Bureau

We are a service oriented business offering discount prices

FAX: 1-214-660-3695

Customer Service and Order Status call: 1-214-988-3500

COMPLETE PRODUCT LISTING AVAILABLE

Call toll free: **1-800-622-9177**

MEMORY EXPANSION

Amiga 500 & 1000

Starboard 0 K	A1000	239.95
Starboard 1 MB	A1000	499.95
Starboard 2 MB	A1000	795.00
Starboard 512 K	A1000	399.95
Starboard Multifunction Card		74.95
Starboard SCSI Module		99.95
Starboard Upper Deck		44.95
M501 0 K		59.95
M501 512 K with Clock		159.95
Mini Meg A1000 with 0.5 MB		359.95
Mini Meg A1000 with 1.0 MB		499.95
Mini Meg A1000 with 2.0 MB		789.95

Amiga 2000 & 2500

8-UP Memorycard DIP	0K	159.95
8-UP Memorycard DIP	2MB	595.00
8-UP Memorycard DIP	4MB	995.00
8-UP Memorycard DIP	6MB	1335.00
8-UP Memorycard DIP	8MB	1795.00

VIDEO SOFTWARE

Artistic Expressions Vol. I & II each	39.95
Broadcast Titler	199.95
Comic Setter	66.95
Deluxe Paint 3.0	109.95
Deluxe Productions	144.95
Director	44.95
Director Toolkit	27.95
Fantavision	39.95
Innvision	169.95
Intro CAD 2.0	49.95
Movie Setter	69.95
Pageflipper Plus FX	99.95
Pagerender 3-D	109.95
Pagestream	139.95
Pagestream Fonts #1-5 each	29.95
Photon Paint 2.0	99.95
Professional Page	239.95
Sculpt 3D	69.95
Sculpt 4D	399.95
TV Show NTSC/PAL	62.95
TV-Text	62.95
Turbo Print	32.95
Turbo Silver	129.95
Zoetrope	99.95
Inovatools I	

MUSIC & AUDIO

Audiomaster II	69.95
CMI Midi Interface	69.95
Dr. T's 4-OP Deluxe	99.95
Dr. T's Copyist	199.95
Dr, T's DX Heaven	99.95
Dr. T's KCS 1.6	166.95
Dr. T's Kawai K1 Lib.	99.95
Dr. T's Midi Recording Studio	44.95
ECE Midi-A500/2000	44.95
ECE Midi-A1000	44.95
"M"	154.95
Midi Cable 5' black	8.95
Midi Cable 8' color	11.95
Midi Gold external	69.95
Midi Gold internal	69.95
Music Mouse	49.95
Perfect Sound	64.95
Soundscape Audiodigitizer	79.95
Pro Midi Studio	124.95
Utilities I	39.95
Studio Magic	69.95
Synthia	[illegible]
Texture	[illegible]

BUSINESS

Analyze 2.0	62.95
BEST Management System 2.0	299.95
Critic's Choice	159.95
Cygnus Ed Prof.	69.95
Excellence	179.95
Financial Plus	199.95
Kindwords	66.95
Maxi Plan 500	99.95
Maxi Plan Plus	129.95
Microfiche Filer	69.95
Microfiche Filer Plus	112.95
Money Mentor 2.0	59.95
Pen Pal	109.95
Phasar 3.0	66.95
Prowrite 2.0	72.95
Super Base Personal II	99.95
Super Base Prof. 3.0	199.95
Superplan	109.95
TexEd Plus	54.95
The Works-Platinum	169.95
Wordperfect	[illegible]
Shakes[illegible]	[illegible]

HARD DRIVES

GVP Impact Drives:	
0MB-80meg Quantum	1199.95
0MB-No Drive	299.95
2MB-80meg Quantum	1699.95
2MB-No Drive	699.95
Hardframe DMA SCSI Controller	239.95
Phoenix Drives:	
22Meg A-1000	599.95
22Meg A-2000	549.95
32Meg A-1000	699.95
32Meg A-2000	749.95
48Meg A-1000	799.95
48Meg A-2000	749.95
80Meg A-1000	1299.95
80Meg A-2000	1249.95
Quantum Pro-80Meg	875.95
Supra 30MB Drive	699.95
Supra 30MB HDisk	699.95
Miniscribe	[illegible]

FDATA-10 DRIVE

[illegible] 3.5" disk drive

FDATA-20 DRIVE

Two 3.5" drives in one small case, clean design, quiet, and at a low price

$ 242.95

HEWLETT PACKARD INK-JET PRINTERS

HP-Deskjet	799.00
HP-Paintjet	999.00

DISK DRIVES

A2010 Internal 3.5"	149.95
AIR Drive External	169.95
AIR Drive Internal	134.95
California Access Drive CA-880	169.95
F-DATA 10	139.95
F-DATA 20	249.95
Drive Extension Cable	19.95

MODEMS

Supra 2400 baud ext.	149.95
Supra 2400 baud int.	149.95

PROGRAMMING

A/C Basic Compiler	129.95
A/C Fortran-AB Soft	179.95
Aztec C Developer	219.95
Aztec C Professional	159.95
Aztec Source Level Debugger	59.95
GFA Basic 3.0	99.95
Lattice C 5.0	219.95
Lattice C ++	399.95
Lint	69.95
Power Windows 2.5	56.95

MISCELLANEOUS

Arexx	32.95
BAD	32.95
CLI Mate	27.95
DOS-2-DOS	42.95
Disk Mechanic	56.95
EZ-Backup	32.95
Facc II	22.95
Project D	32.95
Quarterback	44.95

VIDEO HARDWARE

A-Pro Draw 9x6	399.95
A-Pro Draw 12x12	499.95
Amiga Live A 1000	219.95
Amiga Live A 2000	329.95
Amiga Live A 500	299.95
Color Splitter	89.95
DigiDroid	64.95
Digi View 3.0	139.95
Digi View Gold	139.95
Easyl 500	319.95
Easyl 1000	339.95
Easyl 2000	349.95
Flicker Fixer	499.95
Panasonic 1410 Camera Kit	239.95
Perfect Vision	175.95
Polaroid Palette	2199.95
Super Gen	699.95
X-Specs 3D	99.95

AMIGA 500 Starter Pack

A500 with 1Meg
2 Drives
Monochrome Monitor

949.95

Prepay with check or money order and get free UPS-ground shipping and a FREE GAME!

A-MAX 134.95

The 'MAC' Emulator for the Amiga works with all Amiga models

ROM-Set 139.95

For use with A-MAX

3.5" Drive 199.95

MAC-compatible; for use with A-MAX

TRADE UP!

We will trade in your Amiga 500 or 1000 towards a new Amiga 2000 or 2500. Call for details!

Government, corporate & institutional

Purchase Orders are welcome!

As always we have the most 'customer friendly' terms: S/H $3.95 in cont. USA; Minimum order is $30.00; MASTERCARD & VISA with NO credit card fee; in Texas add 7% Sales Tax. We ship to APO addresses, RMA# required on all returns. 18% restocking charge. All prices subject to change without notice.

Volume 2, Issue 6

Publisher
Antony Jacobson

Vice-President Operations
Jeannie Lawrence

Assistant Advertising Manager
Mike Grantham

Editors
Chris Zamara
Nick Sullivan
Tim Grantham

Assistant Editors
Malcolm O'Brien
Paul Bosacki

Contributing Writers
Steve Ahlstrom
Scott Ballantyne
Richie Bielak
Ben Blish
Daniel Brookshier
Anthony Bryant
Jim Butterfield
Betty Clay
Don Curtis
Matt Dillon
Andy Finkel
Brad Fowles
Eric Giguère
Cathryn Graham
Eric Graham
Chris Gray
Derek Grime
Jay Gross
Eric Haberfellner
Mike Hubbartt
James C. Lawrie
Hobie Orris
Rob Peck
Larry Phillips
Bob Rakosky
Dan Schein
Mike Schwartz
Steve Simpson
John Toebes
Victor A. Wagner
Doug Walker
David Wood

Cover Artist
Anthony Kashinn

for the AMIGA Transactor
The Magazine for Amiga Programmers

Features

AbsExecBase 4
The ethics of 'taking over the machine', and a change in locale for the founding editors.

Programming In M2AMIGA 22
by Anthony Bryant
How to interface the language with shared and scanned libraries.

Creating Amiga Devices 28
by Steve Simpson
This article shows how a 'device' is put together, and how to create your own.

Serial Communications 42
by Mike Schwartz
For *fast* serial communications, you have to go right to the hardware.

Random Number Generation 46
by Eric Giguère
How do you know if a random number generator is good enough?

Colour Cycling 52
by Hobie Orris
Interrupt-driven C routines to generate this powerful animation effect.

Toy: A Small Compiler For The Amiga 56
by Chris Gray
In the third part of the Toy series, Chris examines the compiler's parser in detail.

Introducing Icon 67
by Scott Ballantyne
Icon is a high-level language that excels at string-handling. It's *free* on the Amiga.

Seth: Playing the Chaos Game 72
by Robert P. Hale and Victor A. Wagner, Jr.
A chaos kit for the experimenter. Or is it?

Departments and Columns

Amiga Dispatches 6

by Don Curtis

Hard drive problem-solving techniques, and some notes from the recent Amiga Developer's Conference.

Mastering The Raster 10

by Derek Grime

The Amiga as a graphics rendering machine: does it measure up to the competition?

Assembly Language Programming 14

by Jim Butterfield

Object files, linking, sections, and program units.

NewsLINK 79

Reviews

X-Specs 3D from Haitex 78

Ami...Alignment System from Free Spirit Software 78

About the cover: *Transactor* by Anthony Kashinn

Anthony Kashinn and his wife Laurel are a husband-and-wife computer artist team who come from a background in the more traditional media of watercolour and pen and ink. They are anxious to see the Amiga gain respectability as a fine-art tool, and are dedicated to making their Amiga art studio a full-time occupation. This illustration was drawn in HAM mode using *Digi-Paint*, and is mostly constructed from filled geometric shapes. Using dithering and the 'blend' and 'tint' modes in the program, the shapes were stamped onto the picture to create the subtle smoky, blended look of the background. The 'granite' font from the *Deluxe Paint III* disk was used to create the 'TRANSACTOR' text, and stamped onto the picture various numbers of times in 'blend' mode to get the graduated effect.

TransAmi's phone number is:
(416) 764-5273

FAX: (416) 764-9262

TOLL-FREE ORDER LINE
1-800-248-2719 Extension 911
(orders only; have your VISA or Mastercard number ready; available in the U.S. only)

Transactor for the Amiga is published monthly by Croftward Publishing Inc., 10-85 West Wilmot Street, Richmond Hill, Ontario, L4B 1K7. ISSN# 0838-0163. Canadian Second Class Mail Registration No. 7690, Gateway-Mississauga, Ont. USPS Postmasters: send address changes to: Transactor, PO Box 338, Station C, Buffalo, NY, 14209.

Croftward Publishing Inc. is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names are registered trademarks of Commodore Inc.

Subscriptions: see order card at centre of magazine for complete ordering information and prices.

Send all subscriptions to: TransAmi Subscriptions, 10-85 West Wilmot Street, Richmond Hill, Ontario, Canada, L4B 1K7. For best results, use the postage paid card at the centre of the magazine.

Quantity Orders: In Canada: Disticor, 155 Gordon Baker Road Suite 213, North York, Ontario M2H 3N7. Tel (416) 496-8444 Fax (416) 496-8448. In the USA: IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California, 92121, (619) 481-5928; ask for Dave Buescher.

Editorial contributions: writer's guides are available by writing to the Richmond Hill address above. Write to the editorial department (same address) with article proposals or articles for consideration.

All material accepted becomes the property of Croftward Publishing Inc., except by special arrangement. All material is copyright by Croftward Publishing Inc. Reproduction in any form without permission is in violation of applicable laws. Write to the Richmond Hill address for a writer's guide.

The opinions expressed in contributed articles are not necessarily those of Croftward Publishing Inc. Although accuracy is a major objective, Croftward Publishing Inc. cannot assume liability for errors in articles or programs. Programs listed in Transactor for the Amiga, and/or appearing on Transactor for the Amiga disks, are copyright by Croftward Publishing Inc. and may not be duplicated or distributed without permission.

Transactor for the Amiga is produced using Amigas and the following products:

Professional Page
Deluxe Paint II
Transcript
Digi-View

Final output by Vellum Print & Graphic Services, Inc., Toronto

Printed in Canada by Bowne of Toronto, Inc.

AbsExecBase

Taking over the machine

I must admit, I had to think twice about publishing Mike Schwartz's article on high-speed serial communications (p. 42). Mike suggests that it's sometimes best to bypass the operating system and get right down to the metal. At *TransAmi*, though, we've always advocated proper programming procedure. Multitasking is too important to sacrifice to developers who blithely bulldoze their way through the system because they can't be bothered to learn to use it.

But after rereading Mike's article, I realized there was a bit of knee-jerk in my first reaction to it. His code produces performance that can't be provided by the AmigaDOS serial.device. Commodore has indicated that it will, at some point, optimize the serial device to use the higher transmission speeds; but Mike's routines work *now*. The price: the program hogs the CPU and causes problems for any other program using the serial port. The latter caveat is not terribly important: I would be surprised if anyone runs more than one program using the internal serial port. It may not even be a problem with plug-in serial ports that use their own serial driver software.

Hogging the CPU is another matter: other programs slow noticeably. BUT... it might be worth it. If we at *TransAmi* can send files to the laser printer at speeds up to 57600 bps, Mike's program will save us time and money.

A few commercial software developers, especially those developing animation products, have told me that they would prefer to take over the machine. They say that the buyer of their products would much rather have the extra performance than the multitasking.

I used to bitch about games that took over the machine. But recently, I have been playing *World Class Leader Board*, a good golf simulation from Access. It multitasks fine – but the graphics and the sound eat up so much CHIP RAM I can't run much else anyway. I don't play many games but I expect this situation applies to most of them. Yes, the Super Agnes chip will give us more CHIP RAM; but I suspect that games will tend to expand to fill the RAM available.

So let's not get too sanctimonious about software that interferes with multitasking. Let's make sure that, as Piet Hein says, our halos are not our horizons.

***

The end of an era: By the time you read this, Nick Sullivan and Chris Zamara will be in Victoria, British Columbia – permanently. They have decided to concentrate their efforts on their first love: programming. But don't panic... they will remain on staff as Associate Editors, still editing and writing for every issue of *TransAmi*. Those of us remaining at *Transactor* HQ in Richmond Hill, Ontario will miss them. We wish them well.

***

Those of you with access to USENET can email me at timg@ziebmef.UUCP. If that doesn't work, try ncrcan!ziebmef!timg. You can also contact me electronically on Compuserve (PPN 76703,4353) on the AmigaVendor forum; and on BIX as 'dispatcher'. I will gladly entertain comments, beefs, article ideas... anything to do with the editorial side of the magazine. Subscription or product queries will be relayed to our order department for them to deal with.

So, until next time... exit(0).

Tim Grantham,
Editor

M.A.S.T.
Technical Excellence

"The Matchbox Collection" A POWERHOUSE OF PERIPHERALS.

MASSIVE MASS STORAGE FOR THE AMIGA 500/1000/2000

Maybe you want the unrivalled FLEXIBILITY of TINY TIGER — our UNIQUE standalone SCSI system, compatible with all Amigas.

POWERUSERS may prefer FIREBALL for their Amiga 2000. This true DMA, DRIVE ON A CARD, is designed to burn the competition on **price** and **performance**.

All MAST hard drive systems utilize State Of The Art, 3.5" SCSI drive technology. Both TINY TIGER and FIREBALL include a BUDGET and PERFORMANCE PLUS line of drives.

The BUDGET range comprise Seagate 30 & 50 megabyte drives.

The PERFORMANCE PLUS range feature FUJITSU 3.5" SCSI mechanisms — an indication of the reliability you should expect from this product. Naturally as another quality MAST product, it is SMALL, LOW POWER and covered by our ONE YEAR limited warranty.

PERFORMANCE PLUS drives are available in 45, 90, 135, and 180 megabyte configurations. They are voice coil, fast, and feature their own 24k byte RAM buffer for even faster access. The 45 meg mechanism is SUPERSMALL — at only 1" high it is the same size as our miniscule UNIDRIVE floppy.

TINY TIGER™
THE PURR-FECT SCSI SYSTEM

IF YOU CAN PLUG IN A PRINTER
THEN YOU ALREADY KNOW HOW TO
INSTALL A TINY TIGER
— Simple as that!

★ includes SCSI interface, drive and power supply
—all in one neat little case

★ drive is preformatted with workbench 1.3 installed
—power on and away you go

★ plugs into A500/A1000 and A2000
—great if you upgrade and ideal for moving software from one computer to another

★ connects thru 8 bit printer port
—a natural for SCSI — its 8 bit as well

★ fast — sure is — try this:
copy all of workbench 1.3 from RAM: to RAM: — you can't get much faster. It should take 22.5 secs. To do the same from Tiny Tiger to RAM:=27.6 secs. IN REAL WORLD TESTS — TINY TIGER PERFORMS.
— at lowest price per megabyte

★ printer is passed thru — transparently

★ pass thru for additional SCSI devices

★ TINY TIGER is attached by flexible cable and can be located in a convenient position
— DOES NOT OBSTRUCT USE OF A500 KEYBOARD

★ Bus expansion is left free for other important peripherals
— like memory

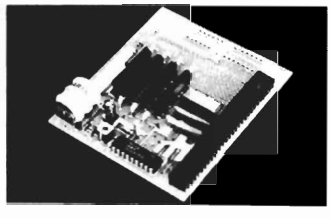
FIREBALL™
HARD DRIVE ON A CARD
FOR THE
AMIGA 2000

★ SCSI INTERFACE AND DRIVE
★ TRUE DMA
★ AUTOBOOT **30 meg to 180 meg**
★ VERY FAST
★ LOW COST
★ INDUSTRY STANDARD 50 pin CONNECTOR ON SCSI PASSTHRU

CAPACITY	FIREBALL	T/TIGER
30 meg	$ 599	$ 629
50 meg	$ 669	$ 699
45 meg	$ 799	$ 799
90 meg	$1099	$1129
135 meg	$1369	$1399
180 meg	$1569	$1599

REMEMBER — T T PLUGS
INTO ALL AMIGAS FOR A
CLEAN, UPGRADE PATH
— DON'T LIMIT YOURSELF
★ BUY TINY TIGER ★

MICROMEGS™ 512 K RAM AND BATTERY BACKED CLOCK



A500 ONLY $169

This alternative to the A501 is only half the size and uses a fraction of the power — important for power conscious A500 owners.

THE ULTIMATE EXTERNAL FLOPPY DRIVES

UNIDRIVE

★ SUPER SLIM LINE
★ VERY LOW POWER
★ NO CLICKING
★ MAY BE SWITCH DISABLED
(TWIN DRIVE HAS TWO SWITCHES)
★ QUALITY FUJITSU MECHANISM
★ POWERED FROM AMIGA™ OR OPTIONALLY AVAILABLE PLUGPAK

UNIDRIVE $159

TWINDRIVE

★ 1 YEAR LIMITED WARRANTY
★ UPGRADE POLICY—BUY A UNIDRIVE AND HAVE IT FACTORY UPGRADED TO A TWINDRIVE ANYTIME
★ REMOVABLE CABLE
★ ROBUST BEIGE METAL CASING
★ HINGED DUST COVER
★ QUIET OPERATION

TWINDRIVE $299

MEMORY TO GO!
NEED MEMORY FOR YOUR A500/A1000
TRY THIS — MINIMEGS™

2 MEGABYTES OF EXTERNAL RAM
in a very small package

NEW LOW PRICES

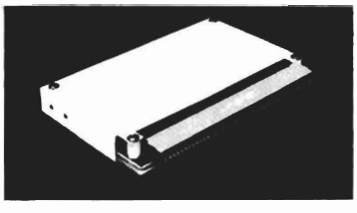
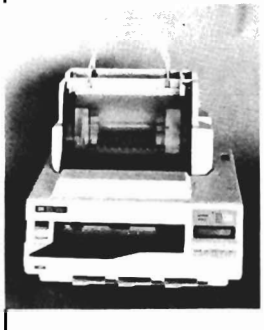


PHOTO OF A500 VERSION

- Auto Configure
- Low Power
- 1 Meg DRAM's
- RAM Tachometer
- Auto Configure LED
- Very Small Size
- Zero Wait States
- Compatible with A501

512K $329 **1 MEG $489**
2 MEG $779 (add $10 for all A1000 versions)

M.A.S.Terprint™ $1695



300 D.P.I.
DESKTOP PAGE
PRINTER

- FUJITSU ENGINE
- LED TECHNOLOGY
  — for increased
  — reliability
- SMALL FOOTPRINT
  —only 16½"x16½"
- DUAL PAPER FEEDER
- HP-PLUS® COMPATIBLE

M.A.S.Terprint PS™ $3495

This is the POSTSCRIPT® version of the page printer. It has 35 fonts and 2 meg of RAM (optional 4 meg). Additional POSTSCRIPT font cartridges can be plugged into the printer.

IMPORTANT: MASTerprint can be upgraded to MASTerprint PS anytime — lets you save up for POSTSCRIPT.

AMIGA 2000 OWNERS
OUR NO-CLICK INTERNAL
DRIVE IS A GREAT
DEAL AT $109

MAST PERFORMANCE PLUS HARD DRIVES AND ALL OUR FLOPPY DRIVES FEATURE FUJITSU MECHANISMS

FROM JAPAN'S LEADING COMPUTER COMPANY

MEMORY & STORAGE TECHNOLOGY INC.

M.A.S.T. U.S.A.
3881-E Benatar Way
Chico, CA 95928
(916) 342-6278 Tel.
(916) 342-6281 FAX

M.A.S.T. U.K.
Unit 1, Dippen
Brodick, Arran
Scotland, KA278RN
(077082) 234

M.A.S.T. AUSTRALIA
178 Pacific Hwy.
St Leonards NSW 2065
(02) 436-2600 Tel.
(02) 437-4919 FAX

Amiga, A500, A501, A1000, A2000, Fujitsu HP+ and Postscript are trademarks of their respective companies. All prices and specifications subject to change without notice.

Amiga Dispatches

A hard drive survival kit, and DevCon notes

by Donald A. Curtis
Copyright © 1989 Donald A. Curtis

Don Curtis is a police officer in Denver, Colorado, who has for the past three years been assigned to program design and development along with system design and maintenance of ten AT & T Unix 3B2 computers. In his spare time, Don is an assistant sysop on CompuServe's Amiga Forums.

A lot of folks are now buying hard drives and installing the drives themselves. So that means it's time for me to pass on a few tips and techniques about working with hard drives.

Unmapped bad blocks on hard drives...

I've had several questions over the last few months from folks who claim to have found a bad block on their hard drive, and even though they go through PREP (or a similar procedure for non-CBM controllers), they can't get rid of it.

The messages usually go something like this, "I've entered the bad block list from the drive during PREP, but I keep getting a key error on key 37934 (or any number) of my drive". The key errors are usually a bad CRC or read/write error. If you keep getting the same key number, you probably do have a block on the drive that's bad, but not on the manufacturer's defect list.

Once they figure out that the block is bad, the next question is how to map that block out to prevent future errors. Well, you could take the drive into a repair shop and have them run it through a test suite, or you could simply figure out where the block is and tell PREP about the new bad block.

Of course, that brings up the question,"Ok, smartie... how do I figure out *where* that bad block is in the form needed by PREP?" Relatively easily... and here's how:

First, a few things you need to know: Key numbers are the absolute disk block (sector) number within that specific filesystem, and they start at number 0 for the first block. That is, if your drive is partitioned into dh1: and dh2:, and presuming they are both large enough, you have a block (or key) 12324 in each of those filesystems.

The second thing you need to know is that the unformatted capacity of a hard disk is not the same as the formatted capacity. And the byte offset you need to give PREP is the number of unformatted bytes from the index mark. Confused? Here's a quick explanation. You probably know that a block in an Amiga filesystem holds 512 bytes.

What you may not know is that besides those 512 bytes in the data block, there are header bytes that tell the filesystem what block number it is. There is a checksum to help insure the data isn't corrupted. By the way, this isn't the checksum the old file system holds in the data block itself, this is a second checksum that's contained in the block header. That is why you'll see hard drive specifications showing both formatted and unformatted bytes: the difference is the number of bytes taken up by the header information.

Next, you need to know that PREP (or equivalent programs) don't really care about the exact byte offset. All they really care about is what block that defect sits in. That way, PREP can re-map that block to another location on the disk and act as though the defect block doesn't even exist.

And last, you need to know that this procedure will *not* work with an embedded SCSI drive. Most of those drives handle bad block mapping by themselves. When most SCSI drives find a bad block, they simply move it somewhere else. No intervention is needed on our part. Consequently, a block numbered 1234 could be anywhere on that partition, whether or not it was a bad block.

SCSI drives like this will report a read error, but should never report a write error.When a SCSI drive attempts to write to a bad block, and finds it can't, it simply writes it to somewhere else. You'll never be the wiser. The controller handles it all. However,

if a SCSI drive is trying to read a block, and after a reasonable number of tries it can't, it will report a read error because it can't determine what data migh have been there.

So, back to our drive problem. The user reports he's getting a key error on key 37934 on partition dh1:. Examining that partition's mountlist entry, we discover that partition dh1: has a LowCyl value of 72, has six Surfaces and 17 BlocksPerTrack. That's all the information we need to know to figure out the proper numbers for PREP. To make this easy, I first figure out where the block is from the very start of the drive, rather than the start of the partition. That makes the math easy (for me) and directly gives me the numbers I need. You could figure out where the bad block was from the start of the partition, and then add in the number of cylinders before the bad block.

Cylinders, like blocks and and heads, are numbered starting at zero. Since this partition starts on cylinder 72, there are 72 'good' cylinders before this partition (numbered 0 through 71). There are six surfaces or heads on the drive (numbered 0 through 5), and 17 blocks per track (numbered 0 through 16).

That means that there are 102 blocks per cylinder (6 times 17). 102 (blocks per cylinder) times 72 (number of 'good' cylinders) equals 7344. The key number we had was 37934 which is the 37935th block, so 37935 plus 7344 equals 45279. That means, from the very start of the drive, the disk defect falls in the 45279th block of the drive.

Divide that number by the number of blocks per cylinder, and you'll have the cylinder number where the defect falls. 45279 divided by 102 equals 443.91. In other words, the defect is just beyond the 443rd cylinder. Since cylinders are numbered starting at zero, the defect is on cylinder 443. Now do you see why I start with the absolute numbers? The number on the left side of the decimal is the number you need to give PREP for the cylinder number! That's *if* there is a remainder. If there is no remainder (if we'd gotten exactly 443), then subtract one from the number you got by dividing. I'll explain why in a bit.

Now subtract the number of good blocks (443 cylinders worth) from our absolute number of 45279. 45279 minus (443 times 102) equals 93. Our defect is in the 93rd block on the cylinder. Since there are 17 blocks per track, we divide the 93 by 17 to get 5.47. Using the same logic as above, the defect is on head 5 (the number to the left of the decimal).

Again, if there is no remainder, subtract one as above. Now, we've got two numbers PREP needs, cylinder 443 and head 5. We still need the byte offset. To get that, we first find out which block on that particular track has the defect. So, we subtract the five 'good' tracks worth of blocks or 5 times 17 from 93. That gives us eight. The defect is in the eighth block.

Now, the eighth block is labeled block 7 (starting numbers at zero rather than one can make life interesting). We still need to figure out how many bytes to use for the byte offset. You'll probably have to take this on faith: a hard drive uses about 600 raw bytes per formatted block. That means that block 0 occupies bytes 0 through about 600, block 1 occupies bytes 601 through about byte 1200 and so forth. We multiply the block number (in this case, block 7) times 600 to get the byte offset at the start of the defective block. That would give us a byte offset of 4200 for our defective block's start.

Now, that value of 600 isn't an exact number; it can vary a bit on either side of 600, though usually not by much. In order to make certain that particular block gets mapped out, we give PREP a number that ends up near the middle of the defective block. Remember, PREP doesn't need the exact number; PREP just needs to be able to figure out which block the defect falls in. To do that, I add 300 (half the number of raw bytes per block), forcing the defect to fall near the middle of the block. This also accounts for any variation from using exactly 600 raw bytes per block. So,we'll use a byte offset of 4500. That gives us all the numbers we'll need: Cylinder 443, Head 5, Byte Offset 4500.

Give PREP those numbers, and key 37934 on partition dh1: will get mapped to another location. Your problems should go away. That's the theory anyway. In practice, I've done this calculation for about five folks, and they've all reported the problem went away. 'Your mileage may vary', but it's worth a try if you think you have an unmapped bad block on your drive.

One thing you should know: when you reformat the drive (and you'll have to after re-PREPing it), don't use the **quick** option and don't be surprised if the drive 'gronks' a bit when it gets to that track. And finally, why subtract one if there's no remainder after division?

Well, let's make it easy: the first partition of the drive starts on cylinder 0 (LowCyl = 0), and we got key 16 error. Figure this out in your head: Since there are 17 blocks per track numbered 0 through 16, the defect is in the last block on the first track (head 0). Key 16 is the 17th block on the drive. The first step would have been 17 divided by 102 equals 0.1666, to give us a Cylinder 0.

Well, that's correct. Our remainder is still 17 and dividing by 17 would have given us 1.00. That would have indicated head 1, and that's wrong! We know the defect is on head 0 because that's where I said it was! So here's the secret: if your division comes up even, you subtract one from the number you got. In this case, we'd end up with head 0 and still have a remainder of 17, the 17th block being where the defect is. The 17th block is block 16. Sixteen times 600 gives us 9600 and adding the 300 would give us an offset of 9900. These numbers are correct: Cylinder 0, Head 0, Byte Offset of 9900.

More on disk drives...

I've seen a relatively large number of complaints about Seagate hard drives on the networks recently. The problem affects several manufacturer's drives (I've had it happen with a Tandon drive), but since Seagate sells so many drives, their name

keeps coming up in the discussions. It appears that some of them will seem to work just fine for a while, and then slowly, they'll start to have problems spinning up on a cold start. If they got started, everything was fine, they worked well – but if you turn it off, it might never work again.

The problem is excess lubricant on those drive platters that are listed as 'plated media'. It happens during assembly: the platters are supposed to get a *single* wipe with a cloth impregnated with a special lubricant. The platters are then subjected to an optical inspection. Many of the platters would fail the optical inspection because of an uneven application of the lubricant. When some of the workers found that the platters they lubricated were failing, they made certain the drives would have an even coating of lubrication by wiping the platters up to five times with the cloth! While the platters would then pass the optical inspection, they wore too heavy a coat of lubrication.

Because of the way the heads 'fly' over the platters, after a time, they force the excess lubricant to form into thin lines known as 'white worms'. By the way, I'm not talking about large gobs of lubricant. They're so small, you have to look at them under magnification. But when dealing with the distance (or lack of it) between the head and the platter, even a smoke particle is 'large'.

These white worms don't cause a problem while the drive is spinning, but do cause a problem when the heads land after the power is turned off. When the heads land, they sink into the lubricant. The meniscus formed around the head acts like glue and prevents the head from moving from the parked position This prevents the drive from spinning up.

You can see this effect yourself. Take two pieces of flat, mirror polished metal. If they're both dry, you can lay one on top of the other and not have any problem taking them apart again. However, if you put a drop of oil (or even water) between the plates and rub them together so that the drop spreads out, you'll find it almost impossible to separate them unless you slide them apart.

I forget the exact name for this phenomenon, but I believe it's called 'molecular adhesion'. Disk drive manufacturers call it 'stiction'. The name doesn't matter. What does matters is that your drive is now useless.

Yes, you can get the drive going again. It's actually easy. You just turn the spindle by hand when powering-up. That will free the heads, and the drive will spin up just fine. Obviously that's not a good idea for a couple of reasons. It's not the way the drive is supposed to work; and you'd have to leave it lying open on your desktop.

Another fix is to have a hard drive repair facility replace the platters. That'll cost you about $100 depending on the drive and the repair facility. Of course, the best fix is to have the drive manufacturer replace the drive. Whether or not the manufacturer will replace the drive depends on the age of the drive, the history of that particular model and manufacturing run, and perhaps how much you complain (politely, please... yelling doesn't help).

And even more on hard drives....

Some SCSI hard drives use 'zones' on the drive that have more blocks per track on the outer cylinders than on the inner cylinders. Furthermore, the drive controller reserves one or two blocks per track for bad block mapping. So, how do we describe the drive to PREP, and how do we properly partition it? To put it in a nutshell, the mountlist(s) can 'lie' about the configuration of the drive, but *must* agree with each other as to the absolute sectors involved.

Are you confused again? That's good, so am I. So let's work this out. Take an imaginary drive, one with 1000 cylinders, six heads with the first 250 cylinders having 35 blocks per track, and the last 750 cylinders having 26 blocks per track. First, let's remove the blocks reserved for bad block mapping (this drive reserves one block per track for that purpose). Now we have zones of 34 blocks per track and 25 blocks per track.

If we want the whole drive as one big partition, it's straightforward. We calculate the total number of sectors, and then find a single arrangement of cylinders/heads/blocks per track that will maximize the number of available blocks. Remember from our above discussion that SCSI drives don't care about those values; they only care about the absolute block number. Programs like PREP or Format merely tell the drive to work on *x* number of blocks; the SCSI controller does the work and figures out how to do it.

Here, our total number of available blocks is (6 x 250 x 34) + (6 x 750 x 25) or 163500 blocks. We want to keep the mountlist close to the real configuration of the drive, but again, we don't need to be precise. We can't use 1000 cylinders because that would leave us 163.5 blocks per cylinder. But what we can do is divide that number by the number of heads to give us the number of blocks per cylinder. That works out to 27250 blocks.

We could create a mountlist showing 6 heads, 1 cylinder and 27250 blocks per track, but that's ridiculous. Besides that, cards like the A2090(a) can't handle more than 127 blocks per track. We could say that the drive consists of 125 blocks per track, and end up with 218 cylinders (27250 / 125 = 218); but again, that doesn't accurately reflect the drive's configuration.

The way to work out the number is to figure out what's close, then 'fudge'. We'll say the drive has 1000 cylinders, six heads and 27 blocks per track. That gives us a total block count of 162000. We are 1500 blocks short of the total blocks that are really available (163500), so we add cylinders to make up the loss. Since we're using 27 blocks per track, and six heads, that's 162 blocks per cylinder. 1500 blocks divided by 162 equals 9.26 extra cylinders. So, if we tell the system that the drive has six heads, 27 blocks per track and 1009 cylinders, our total is 163458 blocks or only two short of the actual number!

I could live with that small a loss, only 1K of data storage. With those numbers, you can also partition the drive, but you must describe the drive the same way in *all* mountlists dealing with that drive. That is, they all must show six heads and 27 blocks per track; the only difference should be the LowCyl and HighCyl numbers.

If we had tried to describe the drive using the original zone values, that is, the first partition having 6 heads, 250 cylinders and 34 blocks per track, with a LowCyl of 0 and HighCyl of 249, it would format fine.

However, the mountlist for the second partition, showing six heads, 750 cylinders and 25 blocks per track starting at cylinder 250, would over-write part of the first partition. The software assumes that the entire drive matches the description given in the mountlist and determines the start block based on that description.

In this case, the software multiplies 6 times 250 times 25 to figure out the starting sector. But, the first partition would start at block 0 and go through block 50999. Consequently, formatting starts at block #37499, over-writing the last 13500 blocks of your first partition!

As you can see, you must use a lot of care when describing a zoned SCSI drive to the system. If you get it wrong, you will have problems and/or lose a lot of the data storage available to you on the drive. The best way to ensure that things work the way you think they'll work is to make sure you get the manufacturer's spec sheet for your drive.

Once you have that, sit down and go through the calculations to figure out how to describe the drive. Figure out the absolute starting and ending block numbers for each partition based on the mountlist entry you intend to use. Make sure they neither overlap nor waste space.

Finally, a few notes on DevCon...

Every year, Commodore-Amiga holds a developer's conference known as DevCon. It's at DevCon that the Amiga hardware and software engineers can get together with software developers and exchange ideas. Future directions of the Amiga are discussed, as are possible future products.

This way, developers get a good idea of what changes are coming, and can take them into account when designing programs. The Amiga engineers also get to hear about the developers' and users' gripes/wishes about the Amiga.

One of the things that users want is virtual memory. A quickie explanation of virtual memory: a method of configuring RAM as a resource limited only by the capacity of external storage. This is normally accomplished by swapping, or moving, data out to a temporary storage device while it's not needed. This frees RAM for data that is needed at that particular time. The data is usually swapped to what is commonly known as the swap partition on the hard drive.

Of course, using a hard drive as 'memory' will slow things down significantly, but intelligent schemes that determine what data is needed and not needed, will minimize the slowdown. If you've enough real RAM to run all your programs at once, then swapping will never occur.

Another topic at DevCon was file links. That is, a single copy of a file on your drive can be 'located' in several places or under several names via linking. One very simple benefit of this feature is the elimination of duplicate code. A single program can do different tasks depending onits name.

For example, you could have a program called 'Harry' and link the name 'George' to it. You will only have one copy of the code, but two file headers, one for each name. With links, Harry would get called by the name George. Since part of invoking a program includes passing the command line to the program, the program can very easily tell who called it. This is much better than the current alias capability of the new shell. When you type the alias 'George' at a CLI prompt, the shell converts it to 'Harry' and Harry gets called by that name.

You can also use linking to speed searches for files. Rather than adding a bunch of directories to your search PATH, or putting all your executables into a single directory, you simply link the executables to your C: directory. Again, the files themselves may actually be spread over a large number of directories; but if you look in C:, you'll find what looks like a copy of the file. In reality, all you have is a pointer to the actual file rather than a duplicate of the actual file.

These are very simple examples. Suffice it to say that linking is a good thing and has many more uses than above.

If all goes well, we may have both virtual memory and file linking when version 1.4 of the OS comes out. Plus other goodies. Things like uniform file requesters, more video modes, and a redesigned and friendlier WorkBench.

Remember though, very little in 1.4 is 'locked' in yet. As Commodore put it, just about everything is still open for discussion so no guarantees. One thing is certain. Version 1.4 of the Operating System will come standard with a full implementation of William Hawes' AREXX Language System. This is a very powerful addition to the system software, and will give you, the user, much more control over how you use your Amiga. For more information on what AREXX can provide, read Ben Blish's article *Implementing An AREXX Interface* in *Transactor for the Amiga*, Volume 1, Issue 6.

No dates have been given for release of 1.4, and I presume it's release will be handled in the same manner as 1.3. That is, it'll be available 'when it's ready', whenever that may be. So don't get discouraged or upset with Commodore if you don't see it until sometime in 1990. Version 1.4 is a major effort and will require a ROM change. Commodore isn't going to start to burn ROMs until they're positive the software is right. Believe me, it will be well worth the wait.

Mastering The Raster

Can the Amiga survive in the desktop video market?

by Derek Grime

Derek Grime is president of Beyond Graphics, a Toronto-based studio that uses the Amiga for a variety of graphics, animation, and video applications. In this regular column, Derek talks about using the Amiga for professional work. Derek can be contacted at Beyond Graphics, P.O. Box 1249, Station F, Toronto, Ontario, Canada, M4Y 2V8.

Let's take a trip back in time to 1985. Commodore had just introduced the Amiga 1000 and there was excitement in the air. You can imagine the impact that this magic box had on the computer-aware public. Here was a graphics powerhouse at an affordable price. Finally there was a machine designed for the visual artist, the animator, the video professional. In 1985 the Amiga had the same capabilities as machines costing $30,000 and more.

Two types of creative users began snapping up Amigas. They were the artists with hacker mentalities and the hackers with artist mentalities. Their numbers were small. Most of the best work they produced came from intimate meetings at video studios, arts collectives, universities and other organizations open to new forms of self expression. These underground discussions directly gave birth to the new field of desktop video.

It was all made possible by a small RCA jack on the back of the 1000. The built in 'video out' enabled the artist to dump his or her work directly to video tape. It was a revolutionary concept. For the first time, moving visual creations could be traded among individuals. Programs like Aegis Animator allowed simple music videos to be completed in a couple of hours. Early user group meetings often ended with tapes of the latest videos being shown.

The competition coming on strong

For a long time the Amiga was the one and only desktop video system. It took two and a half years before the competition began to realize that something was going on in Amiga land. IBM and Apple boxes now offer video support as expensive options. This means that our desktop video market is in for some major changes.

At one time, video output directly from a computer was thought to have only one purpose: it allowed the user to view the data on an affordable television set, rather than an expensive colour monitor. But this capability opened the door to other, more consequential applications.

More than half the homes in North America have a VCR. Home video has boomed like no other new technology. Even feature films often make more money from the home market than they do from a theatrical run.

Videotape makes perfect sense as a mass storage device. VCRs are available for two hundred dollars. For another five bucks the user can record two hours or more of information – information that moves in full colour with sound.

Compare this to the field of desktop publishing. Here, the output device is in the two thousand dollar range; after hours of inputting you are rewarded with a short black and white pamphlet or flyer. This makes perfect sense to the business person but little sense to the home user. To put it another way: how many people prefer reading to watching television? The one-eyed monster wins again.

Despite some marketing problems, Commodore and many software houses have long recognized video as one of the Amiga's great strengths. The other companies couldn't even see this market... until now. Apple has become heavily involved in the field – they are pouring great amounts of money into research and development of three dimensional graphics and interactive video disks. IBM continues to focus its work on the business presentation area. Its latest hardware provides advanced graphics capabilities now.

Where we need to catch up

Let's start with the bad news. There are some real advantages being offered by the other platforms. Here's what to be aware of and wary of.

Display colour is an area where the Amiga is losing ground. The competition offers a total palette that contains more than sixteen million colours. (Our eye can detect differences in tone up to about four million colours – more bit-planes are not necessary.) To illustrate that massive range in different terms, sixteen million colours allows the artist to do a gradation of over four thousand values between every consecutive colour in the

Amiga's palette. At this point, pixels become paint-like and pleasing to the eye.

There is a major flaw, however, in other computers' display capabilities: they only allow the artist to choose two hundred and fifty-six colours from the total palette. A graphic of a cloudy sky can easily use up all the colours available and have none left over for other details. Of course the sky will have an almost photographic quality that we just can't obtain, even in HAM mode. And there are lots of software tricks like dithering and image processing that can give the appearance of more than the two hundred and fifty-six limit.

In many cases, however, no software tricks are necessary. For example, a typical titling job still leaves hundreds of colours left over for the important task of anti-aliasing the image.

It sounds serious, but all is not lost on the colour front. Several companies, including Commodore, are hard at work on colour cards for the Amiga that will offer us this many colours and more. Advertisements for these products have even appeared in some publications – which does not necessarly mean that they are close to release.

Colour cards are not cheap. They can be produced for as little as a couple of thousand dollars; colour/genlock/video cards on the Mac II can cost over ten thousand dollars. Some Amiga developers have confided in me that they don't know if our market can bear the high cost of the equipment. Retail prices could be in the four to five thousand dollar area.

I feel it's a small price to pay for the results. A complete Amiga 2500 with genlock, colour card, software, extra ram and a hard drive could retail for less than ten thousand dollars. Compare this to a loaded Mac II system: a quote that we received in our daily junk mail was for twenty-three thousand dollars without software. Makes the price of the Amiga system look pretty reasonable, doesn't it?

Putting it in real world perspective is even more shocking. A professional paint box is not considered expensive if it retails for fifty thousand dollars. The Amiga could be the world's first affordable (to the professional) paint box computer.

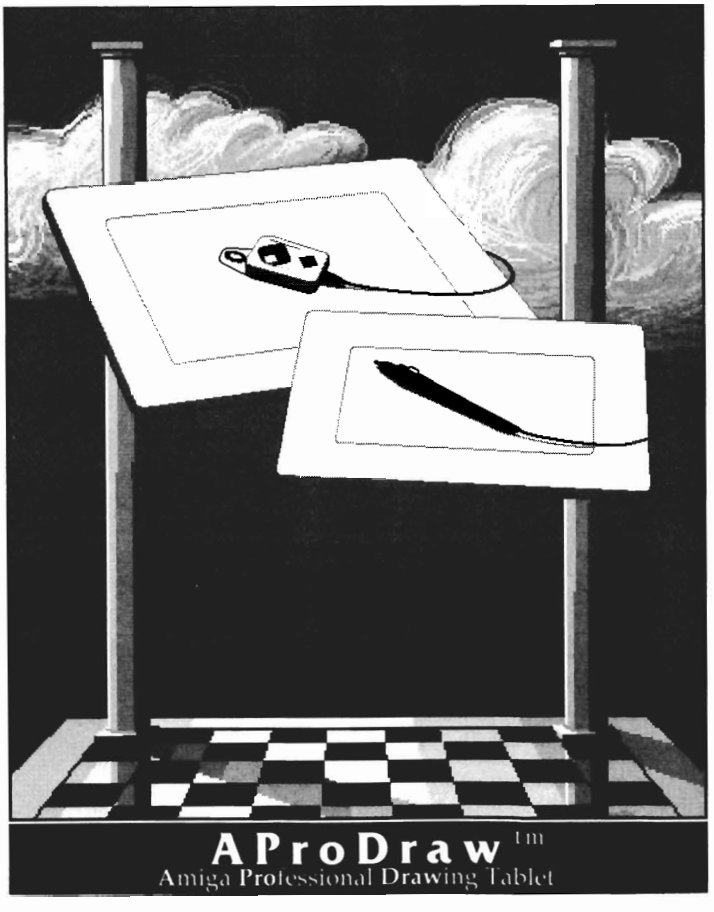
Write to developers if this is a product you would like to see on the Amiga. They need to hear from the public.

3D or not 3D, that is the question

The Amiga was the first affordable home computer to make tracing rays possible. The ray-tracing hobbyist can create a world, set the stage, populate it, light it and wait for the results. For the non-artist, it offers the ability to render realistic graphics without having to create them by hand.

But Ray tracing on the Amiga makes no sense to the professional animator. (If you are a fan of ray-traced, three-dimensional animation on the Amiga, prepare to be offended. Maybe you should stop reading now and skip to the next subhead.)

. . . if you have at least 3 years of assembly language programming experience, preferably one year of 68000 or have an NTSC/PAL Video Hardware background. Join the team that makes no compromises and help us change the world. Key positions available at the fastest growing Amiga company. Hackers call Stephanie at 913-354-1146 or write to Stephanie at 115 West Crane Street, Topeka KS 66603.



Ray-tracing is the current big buzzword in the Amiga market. Videos have been prepared that promote the Amiga as a rendering machine. After a particularly impressive demo, a grim faced announcer offers the following weak caveat: "Be warned this animation uses lots of memory, sometimes more than a megabyte."

Come on guys, what about the time factor? The calculations necessary to produce a ray-traced image of any value are staggering. In animation many frames are required for even a short sequence. A scene can tie up the machine for weeks. To see why, let's outfit ourselves with the best possible equipment, and add up the numbers.

As a hardware base we'll choose a 2500. The 68020 with a math co-processor can render roughly ten times faster than a stock 68000 based Amiga. For software you can't do better than Byte by Byte's Sculpt/Animate 4D. It has a superb animation editor.

Now for the animation. Our hypothetical test will be a logo spin since that seems to be what most 3D clients want. The scene required is only three seconds long. Many studios render sixty frames for every second of screen time. This yields the smoothest possible motion. Thirty frames per second is more than enough in almost every case. At this frame rate, every rendered picture is repeated twice. Some studios push it even further and use only fifteen frames per second. While this is common practice with traditional cel animation, the 3D counter part becomes jerky and shaky when you repeat every image four times.

We'll stick with thirty frames per second. This gives us a total count of ninety ray-traced images to fill three seconds. Phew! You can see how rendering speed becomes a huge issue right away.

We can optimize some things before we start. Keeping the lights, dithering and total polygons to a minimum will give us quite a bit of extra speed. We have to use HAM of course, but anti-aliasing can be kept to a low level. We can make an optimistic guess of thirty minutes per image. Real world results would vary wildly but we are stacking the deck in our favour. We won't factor in gruesome events like power outages or software failures.

This means a three second animation takes forty-five hours of machine time. Your hourly rate would have to be quite low to stay competitive with the competition. Let's compare our results with their results.

How do we stack up?

Three dimensional animation platforms come in three different flavors. Down at earth level we have the 68000 and 386 series machines. PS2's, Macintosh's and Amiga's all are in this realm. Costs range from a few thousand up to over twenty thousand dollars for a usable configuration.

The Amiga based system listed above is by far your best value in terms of price and performance. Rendering speed is about the same for all systems, and while we have an edge in total price, they have an edge in final image quality. Be aware that they pay much more for software and peripherals than we do, and this makes our position even more secure. The competition also has grave problems with their video encoders, a problem the Amiga has had licked for a long time.

The next tier starts at forty thousand dollars. For that price you get an IBM that even it's own mother wouldn't recognize. A blue box forms the base unit under an array of rendering accelerators, twenty-four bit plane colour, high-quality video output and custom software. Rendering speed is fair, and our example job could be completed over night.

The next level takes us into the stratosphere. In the rarified air over fifty thousand dollars we find computers that can handle three dimensional visualization in real time – workstations with accelerators, parallel processors or RISC (reduced instruction set) CPUs. You pay heavily for this much power. Hardware prices average at around two hundred thousand dollars.

Software begins at ten thousand, averages around fifty, with the better known packages coming in around the hundred and fifty thousand dollar mark. And you thought Calligari was expensive! Even more surprising, this only allows the user a one year licence on the product. After that the software quietly deletes itself from the hard drive if you haven't renewed. Let's hope this type of software doesn't become popular on the Amiga.

Where do we go from here?

There is a revolution happening in the 3D world right now. Workstations have been dropping in price. A Sun Sparcstation is available now in a basic configuration for just over ten thousand dollars. A system like this is far more powerful and less expensive than the turbo charged PC platforms.

The middle ground is being nibbled away at both ends. Amigas will inherit one end of the spectrum while the new, cheaper workstations will take the high end. The market is open for us now, but only if a colour card is available in the near future. Our four thousand colour palette is suited for some types of 3D rendering and animation, but not for ray-tracing.

In Beyond Graphics' four years of doing Amiga video we have yet to do one commercial job that involved ray-traced animation. For this article I phoned some of the other Amiga studios around town and heard the same thing. This is not to say Amiga studios don't do professional 3D work. Next month we'll examine the work that is done and why our market niche is just as solid as it has ever been.

Until then why not write and share with us the type of video work that your business specializes in. As we examine our markets further, real world examples could show the range of quality work being produced on the Amiga. We will also look at why the Amiga is becoming a standard for this type of work and what is coming for us in the near future. Recent turbulence in the computer and video market is actually working in the Amiga's favour.

Hold on until next month, the best is yet to come.

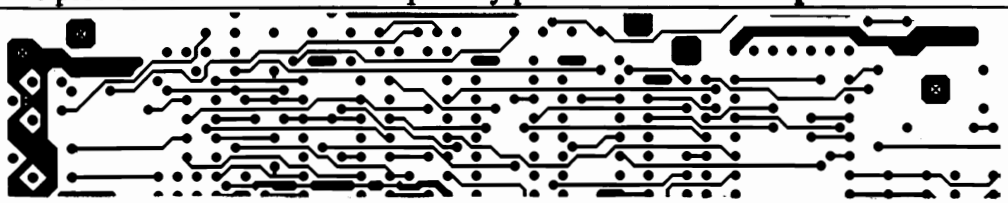
Black Belt Systems

is pleased to present

Board Master

Here's a product for the electronic engineer, technician, or hobbyist. Priced at only **$99.95**, this Printed Circuit Board layout program that has features that rival those of mainframe PCB CAD systems. Board Master is *extremely* fast, flexible, and reliable. Need quality output of small boards? We can use our HP Laserjet II to generate plots of PCBs up to 4x5 inches at 2:1 for a small fee. Board Master requires 960 k of free memory to run, so you will need an expanded Amiga to use it. Board Master will run on any model Amiga, the memory it requires is mostly fast ram. Lattice 'C' Language source code for our plot drivers is available. Inquire by phone. **Schematic Capture is next!**

- Excellent ARexx interface
- Over 60 PCB ARexx cmds!
- .050 or .025 grid system
- Complete library facility
- Multiple silk screens
- Generates Excellon Drill tapes
- Creates IFF images of PCBs!
- On-screen help, user definable
- Interactive Trace Router -
  - ★ Diagonal or Rectangular
  - ★ Dynamic Trace Sizing
  - ★ Global Trace following
  - ★ Autorouting w/ARexx
- Export hole sizes to silkscreen
- Auto DIP, SIP, and PINGRID
- On-screen plotter simulation
- Global UnDo function
- Extraordinary Macro Capability
- Pan easily with prop gadgets
- Runs on *any* Hard Disk Drive
- NOT COPY PROTECTED!



Actual LaserJet II Output plot segment - support for HPGL, HIPLOT, HP LJII

Black Belt Systems
398 Johnson Road
RR1, Box 4272
Glasgow, Montana
59230
406-367-5509

MC-Visa, Check, M.O. C.O.D.
-or-
See your Dealer!

☆ ☆ ☆ ReSource ☆ ☆ ☆

Intelligent Interactive Disassembler for Amiga Programmers

"**ReSource** *makes full use of the Amiga windowing environment ... One of the features that distinguish* **ReSource** *is its ability to produce output which is compatible with most popular assemblers ... a bargain in this world of overpriced software." — Andre Marino*

ReSource is an interactive disassembler for the Amiga computer. Written entirely in assembly language, this is the most powerful disassembler available on *any* personal computer! Over 700 menu functions allow you to:

- ✔ Load from *any* file, read tracks, or disassemble memory directly.
- ✔ Save ALL, allowing you to continue disassembling later.
- ✔ Save to a ".asm" file, partial saves supported.
- ✔ Automated symbol creation:

```
JSR      -$1E(A6)    becomes  JSR      _LVOOpen(A6)
MOVE.L   #$3EE,D0    becomes  MOVE.L   #MODE_NEWFILE,D0
```

  Over 5000 symbols supported!
- ✔ Incredibly smooth scrolling.
- ✔ Full key rebinding supported. Load/save keytables.
- ✔ Up to 38 user-defined macros available. Load/save/overlay macros, looping/conditionals supported, variable execution speed.
- ✔ Numeric and text processing functions allow you to automate many things that used to make disassembling so slow!
- ✔ Search function uses full ARP wildcards. Search forwards, backwards, or use the special bi-directional search!
- ✔ The *fastest* text rendering available on the Amiga.
- ✔ Forward and backward referencing allows you to quickly examine referenced code/data, and return to your original position. Nested references supported.
- ✔ Automatically/manually create labels. Manually create/edit/delete labels, symbols, end-of-line comments, full-line comments.
- ✔ Special support for "C" programs, including "A4" register data referencing.

US$75 **If you're serious about disassembling code, look no further!** **A$90**

To order **ReSource V3.05**
Send check or money order
Credit cards accepted

The Puzzle Factory
P.O. Box 986
Veneta, OR 97487
U.S.A.
(503) 935-3709

Glen McDiarmid
28 Marginson St.
Ipswich, QLD
AUSTRALIA 4305
(07) 812-2963

Assembly Language Programming

Object files and linking: Part one

by Jim Butterfield
Copyright © 1989 by Jim Butterfield

Jim Butterfield is a Toronto programmer and author whose articles, books, lectures and television programs have made his name a household word among Commodore users all over the world since the early days of the KIM-1 microcomputer.

We're going to take a leisurely tour of the business of system building – as the professionals do it. We'll look at the idea of building your own library, and how you put your program together in modular chunks.

In the past, we've zipped by object files rather quickly, viewing them simply as an intermediate stage on the way to our target, the loadable file. Now, we will look at object files in more detail. This will involve a surprise visit to AmigaBasic to help us analyze some of the things that are contained within an object file. We'll look to object files as building blocks that can be used to put together our final program masterpiece.

We will also look briefly at program hunks, which let us break our program into pieces that will scatter-load across memory.

All this will be done on a small scale – our space here permits programs of a modest size only. Of course, all of our code could be slammed together into a single source file, assembled and linked as a unit. Such small programs can be debugged without serious difficulty; but big jobs would need to be modularized to help you check out the pieces before staging the whole job. So... although you would be able to put together these example programs as a single unit of reasonable size, bear with me. We'll gain some insights and learn some program building techniques that will be useful on more ambitious jobs.

We will need to be careful with Amiga terminology. You've already noticed multiple names before: Workbench drawers are called *directories* by the CLI. When we talk about program makeup, hunks are sometimes called *segments*, at other times *sections*. Libraries such as 'Exec library' will be more precisely called *resident libraries* to distinguish them from the libraries searched by the linker (such as *amiga.lib* or *small.lib*), which are called *scanned libraries*. Terminology gets picky and tricky here, and not even the official documentation gets it right all the time.

Program hunks, memory segments and coding sections

The programs we have written so far have been single hunks of code. When we type the program name, the Amiga loader finds an empty place of the right size (a *segment*) in memory, and then brings the program in. But a program doesn't have to be set up in this simple way. It could be composed of a number of hunks. When the loader spots this, it looks for several segments of appropriate sizes, and brings in each hunk to its allocated segment. The segments don't need to be close to each other in memory. As the hunks load, the loader program will make the necessary connections between them.

Why bother? Why not make every program into a single hunk of whatever size it needs? There are two technical reasons.

First, some parts of the program may need to go into certain kinds of memory. You may know of CHIP memory, which is an area of memory that can be reached directly by Amiga's special-purpose chips. These chips (Paula, Agnes and Denise) have many special functions, and are especially talented at graphics and sound. If a program contains a graphics image or a sound waveform, for example, it may prove useful to direct the loader to put this information directly into CHIP memory, where the custom chips can access it. Other parts of the program, such as instructions, don't need to go into CHIP memory; the loader can put them wherever is convenient. The hunk system allows a program to be modularized in this fashion.

Second, breaking a program into pieces allows it to fit more easily into the computer. A single hunk of 100 Kb would call for the loader to find 100 Kb of contiguous memory. The same program, broken into ten hunks of 10 Kb each, might be easier to fit into a nearly-full machine.

From a programming style-and-mechanics standpoint, the hunk idea makes sense. A branch or PC-offset address will typically reach about 32000 bytes from the instruction that uses it. If your program uses a block of memory bigger than this, the simpler addressing methods may not work: you might need to use more clumsy code. For this reason, plus the idea that it's good practice to break up your code into *modules*, it's usually quite convenient to program in hunks.

How do you make a hunk? Easy: you use the assembler directive SECTION followed by a name for this hunk.

The SECTION command

Each hunk should start with a SECTION command. A typical command would look like this:

```
SECTION MyProgram,CODE
```

The name – in this example, *MyProgram* – may be anything convenient. Choose distinct names for each hunk, or your assembler might think that you're picking up and continuing a previous hunk with the same name.

The type is shown as CODE, which is the 'usual' type; so much so, that you can omit the type designation and CODE will be assumed. There are two other types: DATA and BSS. They are seldom used in assembly language work. DATA is intended to signify that this section will contain only fixed data information: messages, constant values or similar items. But you don't have to use a DATA section for this kind of information. It may be put into any CODE section, as we have been doing in our programs so far.

BSS signifies 'uninitialized data'. Programmers would call this a work area; it's empty until your running program puts something there. Because a BSS section is empty memory, nothing is loaded when this kind of hunk is encountered; the Amiga loader just sets aside a segment of the appropriate size. If you use a BSS hunk, your program is unlikely to be *pure*; for that reason, many programmers prefer not to use BSS hunks, but instead ask for memory by means of a system call, such as Exec's AllocMem().

SECTION causes the assembler to generate separate hunks. The linker, however, can be asked to join hunks together by means of the SMALLCODE option, which merges CODE hunks, and SMALLDATA, which merges DATA hunks.

Program units and IDNT

A *program unit* is most easily described as the object file generated by an assembly run. It may contain several sections (or hunks), but the object file, as generated by a typical assembler, is a single program unit. It's possible to produce an object file that contains more than one program unit, but we don't usually make these with a simple assembly run.

If we plan to create multiple-program-unit object files (and that's what we're going to do next time as a continuation of this exercise), we should name each unit. We do this with the IDNT directive. It looks like this:

```
IDNT RgDmp
```

It looks something like the SECTION directive, in that it supplies a name. But there will be only one IDNT command per assembly source, since only one program unit will be generated. As with SECTION names, you should make the IDNT names unique.

For this issue's project, we'll construct a useful subroutine, assemble it into an object program – and put it away for future use. Most subroutines can't run without a calling program, but this one is unusual: we'll be able to link, run, and test it by itself. In part two, we'll build another subroutine and combine it with the first one as part of a scanned library.

This routine will be called *RegDump*, which will dump the contents of all the processor's registers. Next time, we'll generate *MakeAscii*, which will convert a longword binary value to an ASCII string.

RegDump: the register dump utility

We will produce this program as a separate program unit. Later, we'll combine several program units to make a more complex object file known as an *archive format file*.

First things first: enter the following as file *RegDumper.asm*. In the future, we plan to combine the resulting object file with other program units; so let's give this program unit a name using IDNT:

```
IDNT ShowRegs
```

The choice of name doesn't really matter. It's only there to keep the program unit separate from any others. Now, give the single hunk a name:

```
SECTION DumpRegs,CODE
```

Again, the exact name is not too important. Here comes some code. Note that this program initializes all its resident libraries, so that it could run by itself; it doesn't have to be called. Most subroutines would expect, and need, to be passed data from a calling program; this one is an exception.

```
; hex register dump   Jim Butterfield
; input needed: none
; output supplied: hex dump to standard output
; no registers affected

  XDEF RegDump

; Exec library calls
  _LVOOpenLibrary          EQU -$228
  _LVOCloseLibrary         EQU -$19E
; Dos library calls
  _LVOOutput               EQU -$3C
  _LVOWrite                EQU -$30

  RegDump:
```

Note that RegDump has been made an 'external definition'; this point in the program will be visible to the linker. If the linker were searching for a label called RegDump, it would find it here. It would *not* find any other program label, such as CopyStr, on the next page.

```
  MOVEM.L d0-d7/a0-a7,-(A7)   ; save all registers
```

Amazingly, the stack pointer itself can be saved onto the stack. This **MOVEM** command has placed the contents of all registers onto the stack. Why? We hardly need to save every register. The reason: we can pick them directly from the stack in order to print them. That's much easier than trying to shuffle the register contents around internally. We must, of course, remember to return those 64 bytes to the stack when we're finished.

```
        MOVE.L 4,A6            ; pointer to Exec library
        LEA    DosName(pc),A1; name of Dos library
        MOVEQ  #0,D0           ; any version = 0
        JSR    _LVOOpenLibrary(A6); go find Dos library
        TST.L  D0              ; did we find it?
        BEQ.S  ForgetIt        ; no, give up
        MOVE.L D0,A6           ; yes, make it active
        JSR    _LVOOutput(A6); get ouput handle
        MOVE.L D0,D7           ; put it in D7
        LINK   A5,#-$24        ; grab 24 working bytes
```

We've now established our output path and are ready to do the information dump. Does it seems strange that we have opened the DOS library when the program which calls this subroutine might already have it open?

No worry: with multitasking, a dozen processes might already have the DOS library open. DOS simply says, "OK, here's another one" and increases its count 'opens'. Later, when we close DOS in this subroutine, the count goes back down. But until the count reaches zero, DOS will not close up shop and go away ('be expunged'). In fact, the DOS library, like Exec, is special and will never go away even if nobody happens to be using it.

Note that another 36 bytes have been taken from the stack, plus four to hold the prior contents of register A5. This not only gives us a place to hold working data, it also positions A5 as a good reference point to reach for our stacked register values.

Next, we'll copy our output 'template' to a work area. Later, we'll fill detailed values into parts of this area.

```
        LEA    -$24(A5),A4     ; text work area
        LEA    OutLine(pc),A3; output template
        MOVEQ  #OutLen-1,D0    ; characters to move
CopyStr MOVE.B (A3)+,(A4)+     ; move a character
        DBF    D0,CopyStr      ; count and loop
```

The **DBcc** instruction group are splendid for setting up a fast program loop. In this case, D0 counts down; when D0 passes zero, the loop terminates. Note that D0 needed to be set up with a value that is one *less* than the number of times around the loop.

```
; ready to run through registers
; D2 and D3 will not be disturbed during loop

        LEA    -$24(A5),A4     ; text work area
        MOVE.L A4,D2           ; = output address pointer
        MOVEQ  #OutLen,D3      ; = length of output string
```

Registers D2 and D3 can be preset for the output function. We won't disturb them as we go around the loop, and thus will need to do that much less setup for the call to Write().

```
        MOVEQ  #0,D4           ; start with register 0
RegLoop:
        MOVE.B D4,D5           ; copy the reg number
        OR.B   #$30,D5         ; make it Ascii
        MOVE.B D5,1(A4)        ; stick in the Dn place
        MOVE.B D5,13(A4)       ; stick in the An place
        MOVE.W D4,D5           ; register number
        ASL.W  #2,D5           ; times four bytes each reg
        MOVE.L 4(A5,D5.W),D0 ; get D register contents
        LEA    11(A4),A0       ; end of D=.... hex area
        BSR.S  ConvertHex
        MOVE.L 36(A5,D5.W),D0; get A register contents
        LEA    23(A4),A0       ; end of A=.... hex area
        BSR.S  ConvertHex
```

Our template has been filled in with the appropriate values for one line of print. So we print it.

```
        MOVE.L D7,D1           ; outhandle
        JSR    _LVOWrite(A6)   ; write the line
        ADDQ.W #1,D4           ; next register
        CMP.W  #8,D4           ; all registers done?
        BLT.S  RegLoop         ; nope, loop back
        UNLK   A5              ; give back LINK memory
        MOVE.L A6,A1           ; DOS library pointer
        MOVE.L 4,A6            ; close DOS via Exec
        JSR    _LVOCloseLibrary(A6)
ForgetIt
        MOVEM.L (A7)+,d0-d7/a0-a7; put registers back
        RTS
```

Now let's do the subroutine that converts each register value to printable ASCII characters. When the subroutine is called, register D0 contains the value to be converted, and register A0 says where to put the characters, starting at the right-hand (low-order) end.

```
ConvertHex:
        MOVEQ  #7,D6           ; Eight-1 hex digits
        MOVEQ  #0,D1           ; Clear longword D1
HexLoop MOVE.B D0,D1           ; Copy the value
        LSR.L  #4,D0           ; Move four bits out
        AND.B  #$F,D1          ; Strip those four bits
        OR.B   #$30,D1         ; Make numeric Ascii
        CMP.B  #$39,D1         ; OK if in numeric range
        BLE.S  Numeric
        ADDQ.B #7,D1           ; else bump to A-F range
Numeric MOVE.B D1,-(A0)        ; send it to the template
        DBF    D6,HexLoop      ; go back for another
        RTS

DosName dc.b   'dos.library',0
OutLine dc.b   'Dx=........ Ax=........',$A
OutLen  EQU *-OutLine
        end
```

Save your source file, and then assemble it; but *don't link it.* The name of your object file should be *RegDumper.o.* If you're using DevPac, remember to click on the 'Linkable' box; if you're using Compute!'s ASM68010, do *not* use the -A (autolink) option.

Put your object file away to a disk. Now, you may link it in order to do a quick test. BLink is most people's favorite linker.

```
BLINK RegDumper.o TO RegDumper
```

There are no external labels, so you should not need a LIB (library) field. Assuming everything has been typed in correctly, executable file *RegDumper* will be created, and you are ready for a test run. Type 'REGDUMPER' at the CLI and the sixteen registers should print out neatly.

You may be surprised to find that most of the registers seem to contain data. You do not need to worry about any of them except A0, which contains the pointer to the command tail, and D0, which holds the command tail length. Since you typed in nothing after 'REGDUMPER', other than striking the **RETURN** key, D0 should contain a value of 1, the length of the newline character.

A BCPL Side Trip

I'd like to mention the contents of the A registers briefly. The Amiga loader, which is part of DOS, has been running in a language called BCPL, using an operating system called TRIPOS. You don't need to know either the language or the system. But if you ever try to look at some of the Amiga's CLI commands, which are written in BCPL, it might be a help to know a little about the A registers.

A7, of course, is the stack pointer. A2 points to the *Global Vector*, a huge list of jump addresses. A5 contains the address of a 'standard function call' entry point; and A6, instead of holding a library base, contains a pointer to the 'standard function return' program. When a BCPL function is called, A4 is pre-set to the address of the selected function; this is usually taken from the Global Vector. The call goes to the common start point indicated by A5. As the function runs, it will set A3 to the return address, removing it from its usual place on the stack. Eventually, all functions go to the address indicated in A6, where the things are tidied up before the final return via A3.

You don't need to know this kind of thing to program in assembler language. But if you ever try to muddle your way through some pre-written BCPL code, it's handy to have a little background in this area.

The main job

Let's go back to our first exercise: the "Hello, World!" program. We'll make some small changes in it to demonstrate some of the things we have discussed here.

First, we'll make it a two-hunk program. This serves no practical purpose, since the program is so tiny there's no point in breaking it into two pieces. But doing so will demonstrate how it's done, and show some of the considerations involved.

Second, we'll work in a call to our pre-written, pre-assembled subroutine, RegDump. Here we go: I suggest the source might be called *Hello1.asm.*

We will identify the first hunk with a SECTION directive, giving the arbitrary name *Part1* to this section. Then, a few familiar library definitions.

```
        SECTION Part1,CODE

; Hello program
; Startup sequence, CLI only

; Equates for Exec Library
        _LVOOpenLibrary       EQU -$228
        _LVOCloseLibrary      EQU -$19E
```

Program RegDump will not appear in this assembly source file. Instead, we'll leave a note to the linker, asking it to find RegDump at that time. We do this with XREF, the 'external reference' directive.

```
        XREF    RegDump
; Label RegDump must be found during "link"
```

Our program will start with a call to RegDump. Since subroutine RegDump is likely to be a separate hunk, it might be anywhere in memory. So we cannot use a **BSR** (Branch SubRoutine), which can only reach subroutines within 32K or so. Instead, **JSR** (Jump SubRoutine) will connect to RegDump no matter where it ends up in memory. You may recall that we don't need to open any libraries before calling RegDump; the subroutine will do that for itself.

```
        JSR     RegDump         ; dump regs first thing
```

On to the *Hello* coding. You may recall much of this from part one.

```
        MOVE.L $4,A6            ; Exec base
        LEA    DosName(PC),A1   ; Dos name string pointer
        MOVEQ  #0,D0            ; any version
        JSR    _LVOOpenLibrary(A6)
        MOVE.L D0,A6            ; Dos base pointer
        BEQ.S  Exit             ; Zero, quit
```

The program has done its main library opening job, and we're now ready to call Main to do the work of printing our message. But we can no longer call subroutine Main using command **BSR**. That subroutine will be defined as a separate hunk and might be loaded anywhere. As before, we could not be sure that **BSR**, with its limited reach, would make the link.

```
        JSR     Main            ; Do the job
        MOVE.L  A6,A1           ; Dos base pointer
```

```
        MOVE.L  $4,A6           ; Exec base pointer
        JSR     _LVOCloseLibrary(A6)
```

We will call RegDump again before the program winds up. You might like to guess what will be in some of the registers at this point.

```
Exit:
        JSR     RegDump         ; dump regs last thing
        RTS
```

We must define DosName within this hunk if we want to reach it with PC-offset type addressing. As a general rule, we like program counter offset addressing in the Amiga; it's smaller and does not need any relocation adjustment as it loads.

```
DosName dc.b    'dos.library',0
```

Here comes the second hunk. We start with the SECTION command, giving a different name from that of the first hunk. Incidentally, we don't call RegDump from this hunk, but you might try changing the program to do so if you wish.

```
        SECTION MainJob,CODE
Main:

; Main Job
; or we could try JSR RegDump here

; Equates for Dos library
        _LVOOutput      EQU -$3C
        _LVOWrite       EQU -$30

        JSR     _LVOOutput(A6) ; Get output handle
        MOVE.L  D0,D1          ; Handle to D1
        LEA     Message(PC),A0 ; Address of Message
        MOVE.L  A0,D2          ; .. to D2
        MOVEQ   #14,D3         ; Length of Message
        JSR     _LVOWrite(A6)  ; Send message
        RTS

Message dc.b 'Hello, World!',$0a
        end
```

Save the source file, and then assemble it. Remember to generate an object file; but don't link it yet. The object file should be called Hello1.o.

As with RegDumper, we have defined all our system references (the _LVO definitions) using EQU statements, so we won't need to use library files Amiga.lib or Small.lib to find these.

Object program Hello1 signals the linker that it needs to find an external reference RegDump. And looking back at program RegDumper, we note that it offers an external definition we may use called RegDump. It seems as though the linker might find a match.

There are two ways we can do this job. At the moment, they work exactly the same way and produce the same result. First, try this:

```
BLINK Hello1.o RegDumper.o TO Hello1
```

This says, "Take Hello1.o and RegDumper.o, stick them together, and produce an executable file called Hello1".

Now try the second method:

```
BLINK Hello1.o TO Hello1 LIB RegDumper.o
```

This says, "Take Hello1.o and produce an executable file called Hello1. If you have any unmatched XREF items, look at RegDumper.o; if you find a matching XDEF there, include RegDumper.o as part of the package".

The result is the same, either way. You'll end up with a three-hunk executable file. If you like, try it both ways and run each resulting executable program.

What's the difference? Suppose we went back to our source program Hello1.asm and removed the calls to RegDump; they were there only for testing purposes. We must reassemble to produce a new Hello1.o, of course. Now: the first BLink command above will include RegDumper in the final executable package, even though it isn't used. The second BLink, using RegDumper.o as a library, drops it out of the final program since it isn't needed.

HunkView: an object file scanner

I make no apologies for introducing an AmigaBasic program into an Assembler series. The BASIC code is readable and easy to play with, and I wanted to avoid burying you beneath a massive assembly language program that would be needed to do the same thing.

Program *HunkView* lets you look at a object file (or for that matter, most executable programs). It tells you information about the hunks it sees. Most importantly for our purposes: it shows you the XREFs and XDEFs that remain embedded in object files. These references are resolved and removed by the linker. The program helps you to see them there, and to understand what the linker is doing.

HunkView may be used to look at any program – object or executable. It may give you a better feel for Amiga program architecture. While browsing other people's programs or object files, HunkView might show you other information, such as symbol definitions. Don't worry too much about these; they are there to help the debugging process.

Use HunkView to look at the files we have created in this session. You'll see RegDumper.o offering its external definition to the world; you'll see the two hunks of Hello1.o, one of them asking for a symbol it hasn't got. And Hello1 will lose all these symbols as they are fitted together into an executable package of three hunk.

Stay tuned...

In the next session, we're going to link RegDumper with other subroutines to produce a more realistic scanned library. We'll pause to reflect upon the use and convenience of such pre-built libraries, and compare them with macros and includes, which are other methods of storing 'canned' code. Keep the files you create here; we'll use them again.

Listing 1: *RegDump.asm*

```
IDNT  FirstPart
SECTION DumpRegisters,CODE

; hex register dump     Jim Butterfield
; input needed:  none
; output supplied:  hex dump to standard output
; no registers affected

   XDEF RegDump

; Exec library calls
   _LVOOpenLibrary   EQU -$228
   _LVOCloseLibrary  EQU -$19E

; Dos library calls
   _LVOOutput        EQU -$3C
   _LVOWrite         EQU -$30

RegDump:

   MOVEM.L d0-d7/a0-a7,-(A7)
   MOVE.L  4,A6
   LEA     DosName(pc),A1
   MOVEQ   #0,D0
   JSR     _LVOOpenLibrary(A6)
   TST.L   D0
   BEQ.S     ForgetIt
   MOVE.L  D0,A6
   JSR     _LVOOutput(A6)
   MOVE.L  D0,D7
   LINK    A5,#-$24
   LEA     -$24(A5),A4
   LEA     OutLine(pc),A3
   MOVEQ   #OutLen-1,D0

CopyStr MOVE.B  (A3)+,(A4)+
 DBF     D0,CopyStr          ;word size only

; ready to run through registers
; D2 and D3 will not be disturbed during loop

 LEA     -$24(A5),A4
 MOVE.L  A4,D2               ;address of string
 MOVEQ   #OutLen,D3          ;length of string
;
 MOVEQ   #0,D4               ;start with reg 0

RegLoop:  MOVE.B D4,D5
 OR.B    #$30,D5
 MOVE.B  D5,1(A4)
 MOVE.B  D5,13(A4)
 MOVE.W  D4,D5
 ASL.W   #2,D5
 MOVE.L  4(A5,D5.W),D0
 LEA     11(A4),A0
 BSR.S   ConvertHex
 MOVE.L  36(A5,D5.W),D0
 LEA     23(A4),A0
 BSR.S   ConvertHex
 MOVE.L  D7,D1               ;outhandle
 JSR     _LVOWrite(A6)
 ADDQ.W  #1,D4               ; next register
 CMP.W   #8,D4               ; all registers done?
 BLT.S   RegLoop

 UNLK    A5
 MOVE.L  A6,A1
 MOVE.L  4,A6
 JSR     _LVOCloseLibrary(A6)

ForgetIt MOVEM.L (A7)+,d0-d7/a0-a7
 RTS

ConvertHex:
 MOVEQ   #7,D6
 MOVEQ   #0,D1

HexLoop MOVE.B  D0,D1
 LSR.L   #4,D0
 AND.B   #$F,D1
 OR.B    #$30,D1
 CMP.B   #$39,D1
 BLE.S   Numeric
 ADDQ.B  #7,D1

Numeric MOVE.B D1,-(A0)
 DBF     D6,HexLoop
 RTS

DosName dc.b 'dos.library',0
OutLine dc.b 'Dx=........ Ax=........',$A
OutLen  EQU  *-OutLine

 end
```

Listing 2: *Hello1.asm*

```
SECTION Part1,CODE

   ; Hello program
   ; Startup sequence, CLI only

   ; Equates for Exec Library
   _LVOOpenLibrary    equ   -$228
   _LVOCloseLibrary   equ   -$19E

   XREF RegDump

; Label RegDump must be found during "link"

   JSR    RegDump                ; dump regs first thing
   MOVE.L $4,A6                  ; Exec base
   LEA    DosName(PC),A1         ; Dos name string pointer
   MOVEQ  #0,D0                  ; any version
   JSR    _LVOOpenLibrary(A6)
   MOVE.L D0,A6                  ; Dos base pointer
   BEQ.S  Exit                   ; Zero, quit
   JSR    Main                   ; Do the job
   MOVE.L A6,A1                  ; Dos base pointer
   MOVE.L $4,A6                  ; Exec base pointer
   JSR    _LVOCloseLibrary(A6)

Exit:
   JSR    RegDump                ; dump regs last thing
   RTS

DosName  dc.b  'dos.library',0

 SECTION MainJob,CODE

Main:
   ; Main Job
   ; or we could try JSR RegDump here

   ; Equates for Dos library
_LVOOutput         equ   -$3C
_LVOWrite          equ   -$30
   ;
```

```
   JSR     _LVOOutput(A6)      ; Get output handle
   MOVE.L D0,D1                ; Handle to D1
   LEA     Message(PC),A0      ; Address of Message
   MOVE.L A0,D2                ;  .. to D2
   MOVEQ  #14,D3               ; Length of Message
   JSR     _LVOWrite(A6)       ; Send message
   RTS

Message  dc.b  'Hello, World!',$0a

   end

   ; Command BLINK TEST.O TO TEST
```

Listing 3: *HunkView*, an AmigaBasic program

```
PRINT "Simple Object/Loadable Analyzer .. Jim Butterfield"
DATA "CODE","DATA","BSS"
FOR j=1 TO 3
 READ HunkType$(j)
NEXT j
DATA "ext_symb","ext_def","ext_abs","ext_res"
FOR j=0 TO 3
 READ SymType$(j)
NEXT j
DATA "ext_ref32","ext_common","ext_ref16","ext_ref8"
FOR j=1 TO 4
 READ RefType$(j)
NEXT j
BlokSize%=2500
LINE INPUT "File to analyze? ";Fil$
OPEN Fil$ FOR INPUT AS #1
 RamType%=CVI(CHR$(0)+INPUT$(1,1))
 Magic&=CVL(CHR$(0)+INPUT$(3,1))
 IF Magic&=1011 THEN
  PRINT "An Executable File!"
  NameSize&=CVL(INPUT$(4,1))
  WHILE NameSize&>0 THEN
   HunkData$=INPUT$(4*NameSize&,1)
   GOSUB NameTrim
   PRINT CHR$(34);HunkData$;CHR$(34)
   NameSize&=CVL(INPUT$(4,1))
  WEND
  Hunks&=CVL(INPUT$(4,1))
  PRINT "It has";Hunks&;"hunks"
  HunkData$=INPUT$(4*Hunks&+8,1)
 END IF
 IF Magic&>998 AND Magic&<1015 THEN
  IF Hunks&=0 THEN PRINT "Looks like an OBJECT file..."
  Hunks&=0
  Live%=0
  WHILE Magic&>998 AND Magic&<1015
   IF Magic&=1010 THEN
    PRINT:Live%=0
   ELSE
    IF Live%=0 THEN Live%=1:PRINT "Hunk";Hunks&;:Hunks&=Hunks&+1
   END IF
   IF Magic&<1004 OR Magic&=1009 THEN
    Size&=CVL(INPUT$(4,1))
    Style%=Magic&-1000:IF Style%>3 OR Style%<0 THEN Style%=0
    IF Style%>0 THEN
     PRINT " (";HunkType$(Style%);":";
     IF Size&=0 THEN
      PRINT " Empty)";
     ELSE
      PRINT Size&*4;"bytes)";
     END IF
     IF Magic&=1003 THEN Size&=0
    END IF
    IF Magic&=999 THEN PRINT " UNIT";
    IF Magic&=1000 THEN PRINT " SECTION";
    IF Size&<>0 THEN
     GOSUB DataRead
     IF Magic&=999 OR Magic&=1000 THEN
       GOSUB NameTrim
       PRINT ":";CHR$(34);HunkData$;CHR$(34);" ";
     END IF
     END IF
    END IF
    IF Magic&<=1006 AND Magic&>=1004 THEN
     Size&=CVL(INPUT$(4,1))
     WHILE Size&>0
      Hunk&=CVL(INPUT$(4,1))
      GOSUB DataRead
      Size&=CVL(INPUT$(4,1))
     WEND
    END IF
    IF Magic&=1007 OR Magic&=1008 THEN
     Type%=CVI(CHR$(0)+INPUT$(1,1))
     SymLen&=CVL(CHR$(0)+INPUT$(3,1))
     WHILE SymLen&+Type%<>0
      PRINT
      PRINT "  ";
      IF Type%<4 THEN
       PRINT SymType$(Type%);":";
      ELSEIF Type%>128 AND Type%<133 THEN
       PRINT RefType$(Type%-128);":";
      ELSE
       PRINT "???";":";
      END IF
      HunkData$=INPUT$(4*SymLen&,1)
      GOSUB NameTrim
      PRINT TAB(15);HunkData$;
      IF Type%=130 THEN
       CommSize&=CVL(INPUT$(4,1))
      END IF
      IF Type%>=129 AND Type%<=131 THEN
       RefSize&=CVL(INPUT$(4,1))
       RData$=INPUT$(4*RefSize&,1)
      ELSE
       SymVal&=CVL(INPUT$(4,1))
      END IF
      Type%=CVI(CHR$(0)+INPUT$(1,1))
      SymLen&=CVL(CHR$(0)+INPUT$(3,1))
     WEND
    END IF
     IF EOF(1) THEN
      Magic&=0
     ELSE
      RamType%=CVI(CHR$(0)+INPUT$(1,1))
      Magic&=CVL(CHR$(0)+INPUT$(3,1))
     END IF
   WEND
  ELSE
   PRINT "Not an Object/Executable file."
  END IF
CLOSE 1
PRINT
PRINT Hunks&;"hunks in this file."
END
DataRead:
 Blox%=INT(Size&/BlokSize%)
 Tail%=Size&-Blox%*BlokSize%
 BSize%=BlokSize%
 FOR k=0 TO Blox%
  IF k=Blox% THEN BSize%=Tail%
  HunkData$=INPUT$(4*BSize%,1)
 NEXT k
RETURN
NameTrim:
       WHILE RIGHT$(HunkData$,1)=CHR$(0)
        HunkData$=LEFT$(HunkData$,LEN(HunkData$)-1)
       WEND
RETURN
patch:
OPEN "ram:crapper" FOR OUTPUT AS #2
PRINT#2,MKL$(1004);MKL$(4521);
WHILE NOT EOF(1)
 a$=INPUT$(4,1)
 PRINT#2,a$;
WEND
CLOSE 2
CLOSE 1
```

TxEd *Plus*™

The Text Editor for the Amiga™, *Plus* a whole lot more.

New ! V2.0 Supports FFS. **BlitzDisk** Disk cache, speeds up floppy and hard disk reads up to 2000%.

FastFonts & FunKeys FF, the text speedup, and FunKeys, the hotkey window manipulator.

New ! Version 1.3 **ARP** ARP, the enhanced AmigaDOS™ compatible CLI program replacements. New ASH shell, and complete users manual. Save 70 KBytes on your boot disk with smaller, faster, more powerful replacements for the C: commands.

AREXX Demo version of the AREXX the macro processor.

TxEd *Plus* is a powerful yet simple text editing environment designed to meet all of your text editing needs. It's small, fast, and fully configurable. The AREXX macro language connection makes TxEd *Plus* an ideal foundation for your complete multitasking system.

"As a programming and general-purpose text editor, TxEd Plus sets the standard."
-Warren Block, Sept '88 INFO Magazine

"(TxEd V1.3 is) ...a very good editor and an excellent value."
-Jan&Cliff Kent, Vol. 1 #9 Amazing Computing

"FastFonts / BlitzDisk provides much more than Facc II for a similar price."
-Warren Block, July '88 INFO Magazine

"I found the ARP commands to be smaller, faster, and more powerful than their forerunners."
-Jeff Blume, July '88 AmigaWorld

"Yes, get TxEd if you're editing almost anything on the Amiga."
-Bruce Webster, July '86 BYTE Magazine

"Still the one for me."
-Louis R. Wallace, Nov '88 AmigaWorld

$79.95

Microsmiths, Inc
PO Box 561, Cambridge MA 02140
(617) 354-1224 BIX, PLink: cheath CIS: 76004,1766

MC and Visa accepted. Mass Residents add 5% sales tax.
Amiga and AmigaDOS are trademarks of Commodore-Amiga, Inc

Programming In M2AMIGA

Library extensions and interfaces

by Anthony Bryant
Copyright © 1989 A. Bryant

The term 'Library' is greatly overused within the Amiga computer lexicon, where it is used to describe a collection or group of software routines which, because of their interrelationship, just happen to reside together. An Amiga language development system like M2AMIGA has a great variety of these so-called 'libraries' with which to become familiar.

There are the libraries that actually come with a particular language implementation, and there are supplemental libraries that you, the overly-keen programmer, have been busy making. These are scanned by the linker primarily during the assembly phase of program development. Then there are those shared 'resident' libraries (like 'intuition.library') with which everyone is now so familiar that are resident with the Amiga itself, both ROM-based and disk-based. (And, also, the increasingly popular supplemental disk-based 'libraries' like arp.library).

In addition, there is also the paper library that must be maintained and continuously updated by the already over-worked programmer in order to cross-reference all of the aforementioned software libraries, but that's another story.

Calling all libraries

The busy M2AMIGA programmer need not ordinarily concern himself with the mechanism of IMPORTing these resident libraries. This is, after all, one of the M2AMIGA compiler's features. The Amiga interface libraries (with the suffix '.library') that are required by a program are opened by the compiler automatically. This is so that in the event of a 'program interruption' (M2AMIGA users no longer speak of a *crash*), the run-time system can close them. One of the advantages to automatic opening is that neither the opening nor the testing of its success is left up to the over-worked programmer! This is also in accordance with the Modula-2 philosophy that imported code can be used under all circumstances without having to call up special initialization routines.

Let's consider what goes into the making of a so called 'interface' library module. An extension to the normal definition module heading is required to tell the compiler that it is dealing now with a special case – a shared ROM- or disk-based resident library, as opposed to the normal scanned library. To define a definition module as an interface library module, the name of the library and the required version number is placed in braces after the module name, thus:

```
DEFINITION MODULE DiskFont{"diskfont.library",33};
```

When importing this module, the compiler automatically generates the required Amiga code to open and close this library, using Exec functions OpenLibrary() and CloseLibrary(), and sets up an 'internal' variable for the library address (e.g. an internal diskfontBase). I say 'internal' here because the address of the library contained in this invisible variable is not easily accessible. To do so, one must (within a program context), IMPORT the whole module without qualifiers and then get the address using the module's name, passing it to your own variable, thus:

```
IMPORT DiskFont;
...
...
myDiskFontBase:= ADR(DiskFont);
```

In M2AMIGA, only the 'execBase' variable (see definition module Exec) at absolute address 4 is a visible type of library base variable.

The compiler also generates the code to test the success of opening this library and, if successful, terminates use of this library properly. For example, consider what happens when importing a library definition like the above. The value returned by OpenLibrary() must be assigned to an internal variable of a specific pointer type, like LibraryPtr, or else a plain ADDRESS type. (Pointers are assignment-compatible with type ADDRESS.) This variable must be accessible to the module, so that it can load register A6 before calling a library function, and so that it can call CloseLibrary(). So, the compiler might generate the following bit of equivalent pseudocode (the use of the phrase 'equivalent pseudocode' allows me great leeway here to speculate!).

```
TYPE
   diskfontBase = LibraryPtr; (* or ADDRESS *)
...
...

diskfontBase:= OpenLibrary(ADR("diskfont.library"),33);
Assert(diskfontBase # NIL,
    ADR("Error Opening diskfont.library"));
TermProcedure(CloseLibrary(diskfontBase)); (* approx. *)
```

The Arts runtime procedure, Assert(), checks the result from calling OpenLibrary() and, if unsuccessful, halts the program with the all-too-familiar Abort requester. This is most likely to occur when, for example, the run-time system can't find a disk-based library in the 'libs' directory. Successfully opening this library, though, sets up normal program termination by appending a CloseLibrary() call to the linked list of term procedures. With TermProcedure(), a module can install any number of parameterless cleanup procedures, which are then called up either to handle runtime errors (when a program aborts), or to handle a normal program end. The last TermProcedure() to be entered is the first to be executed. This guarantees that multiple, nested, termination procedures are executed in the correct order. That is, those of deeper modules are not executed before those of the main program. At the end of the program, all TermProcedure()'s are automatically deleted.

Types and constants that need to be visible take up the most space in an interface definition module and contribute greatly to the overall size of the '.sym' files created by the compiler. In some cases these '.sym' files are somewhat huge and redundant if the module imports and uses structures from other library modules. Unfortunately, this is especially true of POINTER TO RECORD types whose very definition includes nested sub-structures of each component! The compiler has no alternative but to place a copy of the whole thing – record and subrecord and sub-sub-record, etc., into the symbol file. However, as pointers are all assignment compatible with type ADDRESS you can fake the compiler, in certain cases of pointer imports, by declaring it as an ADDRESS type, and this can yield more compact symbol files.

Procedures are the next component in an interface definition module. Since M2AMIGA, as we have seen in this series of articles, enables parameters to be passed to procedures via registers, their use in interface libraries is natural (this mechanism was devised especially for the Amiga interface libraries). M2AMIGA also has the reserved word CODE, likewise used mainly in the definition module of an interface library. The reserved word 'CODE' followed by an LVO – LibraryVectorOffset – (always a negative number) stands in place of a procedure body, thus:

```
PROCEDURE OpenDiskFont(textAttr{8}: TextAttrPtr): TextFontPtr;
CODE -30;
```

Any interface library, like 'diskfont.library', must have a minimum of the basic four standard functions – Open, Close, Expunge and ExtFunc – which correspond to the user calls OpenLibrary(), CloseLibrary(), and RemLibrary(). (ExtFunc is reserved.) Since LVOs are six bytes long, the first four standard functions use LVOs of -6, -12, -18, and -24, making the first free user function at an LVO of -30. The statement CODE -30 translates into the following compiler-generated opcode, a single inline assembly instruction for each procedure.

```
JSR -30(A6)
```

When such a procedure is eventually imported, the internal library base, 'diskfontBase' goes with it, ensuring register A6 is correctly primed with the address of the library. Parameters are passed via allocated registers. With each procedure body thus defined with a CODE statement, *no* additional binding 'glue' assembly code is needed! No implementation module is needed, no '.obj' module is generated by the compiler, and no additional time is taken up by the linker. Only a '.sym' module is generated by the compiler, allowing all of the information in the standard Amiga include files to be grouped together into one, albeit sometimes huge, definition module. 'Graphics', for example contains all of the more that a dozen standard include files pertaining to graphics.

Alternate strategy

One of the disadvantages of this mechanism of automatic opening of libraries, is that the user is not able to test if a library has been opened successfully, so a program must be stopped at the first sign of trouble if the run-time system can't locate or open a specific library. This is fine for the ROM-based resident libraries, which you absolutely must not be without, but you may have designed your own disk-based library (as a neat method of keeping together your most often-used routines). Or, perhaps you are contemplating the use of a supplemental library, like 'arp.library', in your program. What if users of your program don't have access to these libraries – do you chastise them? Summarily close the program and exit stage right? Or would you prefer to have the alternative strategy of just dropping those extra features instead of just abandoning everything.

When designing a library interface for your own library, it may be advantageous to have it be linked in like a normal scanned library. You could design a normal definition module for the library procedures, including all of the types and constants needed, and perhaps including a visible library base variable as well (otherwise the library base variable remains invisible within the implementation module).

In the implementation module, besides your own function procedures, you would also need your own OpenLib() and CloseLib() procedures, which you could then design to handle the case of 'the missing library', perhaps allowing the user some flexible options. Each of your own function procedures would also require some binding 'glue' assembly code for each of the routines contained in the library, so you also need to make note of the LVO (LibraryVectorOffset) values and register parameter allocations for each of the library routines.

Consider a hypothetical disk-based library called 'extended.library' which you have added to your software library collection. Here is the skeleton outline of the definition and implementation modules needed.

```
DEFINITION MODULE ExtendedLib;  (* A Skeleton Outline *)

FROM SYSTEM IMPORT (* whatever is needed *)

CONST
  (* add whatever constants need to be visible *)
TYPE
  (* add whatever types need to be visible *)
VAR
  extendedBase: LibraryPtr;  (* a visible libBase pointer! *)

PROCEDURE OpenExtLib;          (* a call to OpenLibrary()  *)
PROCEDURE CloseExtLib;         (* a call to CloseLibrary() *)
PROCEDURE FunctionNo1(...);    (* the first lib function   *)
  ...
PROCEDURE FunctionNoX(...);    (* the last lib function    *)

END ExtendedLib.def;


IMPLEMENTATION MODULE ExtendedLib;  (* A Skeleton Outline *)

FROM SYSTEM IMPORT
  INLINE,SETREG,REG;  (* and whatever else is needed *)

FROM Exec IMPORT
  OpenLibrary, CloseLibrary;


(* Here is our cleanup routine
 *)
PROCEDURE CloseExtLib;
  BEGIN
    IF extendedBase # NIL THEN CloseLibrary(ExtendedBase); END;
END CloseExtLib;

(* Make certain the library is open or DO SOMETHING ELSE!
 *)
PROCEDURE OpenExtLib;
  BEGIN
    extendedBase:= OpenLibrary(ADR("extended.library"),0);
    (* rather than use Assert() we do the checking ourselves *)
    IF extendedBase = NIL THEN
      (* NO "extended.library" found, so DO SOMETHING ELSE! *)
      MyAlternativeStrategy();
    ELSE
      TermProcedure(CloseExtLib); (* arrange to leave a tidy Amiga! *)
    END;
  END OpenExtLib;

(* Some assembly language required *)
CONST
  jsrA6 = 4EAEH; pushA6 = 2F0EH; popA6 = 2C5FH;

(* FunctionNo1 - two parameters passed, one return, LVO = -30
 *)
PROCEDURE FunctionNo1(param1, param2: LONGINT): LONGINT;
  BEGIN
    INLINE(pushA6);              (* MOVE.L  A6, -(A7)         *)
    SETREG(0,param1);            (* MOVE.L  param1, D0        *)
    SETREG(1,param2);            (* MOVE.L  param2, D1        *)
    SETREG(14,extendedBase);     (* MOVE.L  extendedBase, A6  *)
    INLINE(jsrA6,-30);           (* JSR     -30(A6)           *)
    INLINE(popA6);               (* MOVE.L  (A7)+, A6         *)
    RETURN REG(0);               (* return value is in D0 !   *)
  END FunctionNo1;
...
...    (* more of the same *)
...

PROCEDURE FunctionNoX(...);
  ...

END FunctionNoX;

END ExtendedLib.imp
```

This alternative strategy obviously requires the generation of both a '.sym' symbol file and an '.obj' object file for the linker, and the above skeleton outline is only one solution of many possible. It could also be used to extract only a few needed library functions from a huge library, where you don't want to take the trouble to write out a complete definition. It may provide a little extra flexibility for those special jobs.

* * *

Correction to Volume 2, Issue 4:

Listing 2 at the end of the article (bottom of page 59) was supposed to be *warptext.imp*, but was actually the same program as in Listing 1, *warptext.def*. The correct progrm can be found on the disk for Volume 2 Issue 4; it is also listed below:

WarpText.imp:

```
IMPLEMENTATION MODULE WarpText;    (* WarpText 1.0 *)

(*
 * "The WarpText code, and the name 'WarpText', is
 *    Copyright 1987 by Bill W. Kelly"
 *
 *  M2Amiga adaptation by Anthony Bryant
 *)

FROM SYSTEM IMPORT
 ADDRESS,INLINE,SETREG,REG;

FROM Graphics IMPORT
 BitMapPtr, TextFontPtr;


(* WarpInfo structure NEEDS the following fields initialized:
 *
 *   textFont:    TextFontPtr to an open font.  (i.e. from OpenFont() )
 *
 *   bitMap:      BitMapPtr to an initialized, 'working' BitMap structure.
 *                   (i.e. Open a window.  Get the pointer to the RastPort
 *                   out of rPort field of the Window structure.  From there
 *                   the BitMap structure is at bitMap field.
 *
```

```
 *   whichPlane:  Which bitplane you want the routine to draw into.
 *                   Numbering begins at zero.  On the Workbench screen
 *                   (or any two-bitplane screen) the ONLY possible
 *                   numbers for  whichPlane  are 0 and 1.
 *
 *   left:        Left edge of the 'window' in character locations.
 *   top:         Top edge of the 'window' in character locations.
 *   width:       Width of 'window' also in character locations.
 *   height:      Height of 'window' -- guess what? Character locations.
 *
 * NOTE: These positions are given in character locations, (columns and rows)
 *       rather than pixel locations, which are relative to the 'window' you
 *       have described to InitWarpInfo().   Position 0,0 is the character in
 *       the top left corner of the 'window'  (which may overwrite the title
 *       and borders, unless you use a "superBitMap" window).
 *       Use left, top, width and height to define a "container" into which
 *       WarpText will pour text (with line end "wrap-a-round" if no LF).
 *)


(* REGISTER USAGE:- InitWarpInfo
 *
 * A0 - Pointer to WarpInfo structure with above fields initialized (wInfo)
 * A1 - Address of top of bitplane, windowTop, etc.
 * A2 - Scratch
 *
 * D0 - Left, Top, Width, Height, etc.
 * D1 - WarpInfo modulo
 * D2 - Scratch
 * D3 - Y size of font
 * D4 - Scratch
 *)
PROCEDURE InitWarpInfo(wInfo{8}: WarpInfoPtr);
 (* $E- no entry/exit code *)
 BEGIN INLINE(
  48E7H,0F860H,      (* MOVEM.L D0-D4/A1-A2, -(A7) ; push these registers    *)

  7800H,             (* MOVEQ   #0, D4        ; clear scratch register D4   *)
  3144H,001EH,       (* MOVE.W  D4, 30(a0)    ; store 0 in curX field       *)
  3168H,000EH,0020H, (* MOVE.W  14(A0), 32(A0)   ; copy width into lastX    *)
  0468H,0001H,0020H, (* SUBI.W  #1, 32(A0)    ; make lastX zero-relative    *)

  2268H,0004H,       (* MOVE.L  4(A0), A1     ; A1 is index into bitMap     *)
  7400H,             (* MOVEQ   #0, D2                                      *)
  3429H,0000H,       (* MOVE.W  0(A1), D2     ; get bytesPerRow into D2     *)
  3142H,0022H,       (* MOVE.W  D2, 34(A0)    ; store in bPMod field        *)

  3828H,0008H,       (* MOVE.W  8(A0), D4     ; get whichPlane into D4      *)
  0E584H,            (* ASL.L   #2, D4        ; multiply by four            *)
  2271H,4808H,       (* MOVE.L  8(A1, D4.L), A1  ; get address of planes    *)

  2468H,0000H,       (* MOVE.L  0(A0), A2     ; A2 is index into textFont   *)
  7600H,             (* MOVEQ   #0, D3                                      *)
  362AH,0014H,       (* MOVE.W  20(A2), D3    ; get ySize of font           *)

  7200H,             (* MOVEQ   #0, D1                                      *)
  2202H,             (* MOVE.L  D2, D1        ; copy bytesPerRow to D1      *)
  0C2C3H,            (* MULU    D3, D1        ; mult ySize * bytesPerRow    *)
  3141H,0024H,       (* MOVE.W  D1, 36(A0)    ; store in modulo field       *)

  7800H,             (* MOVEQ   #0, D4                                      *)
  2801H,             (* MOVE.L  D1, D4        ; copy modulo to scratch      *)
  0C8E8H,000CH,      (* MULU    12(A0), D4    ; mult by top (Y offset)      *)
  0D868H,000AH,      (* ADD.W   10(A0), D4    ; add left (X offset)         *)

  2449H,             (* MOVE.L  A1, A2        ; planes addr to scratch      *)
  0D5C4H,            (* ADDA.L  D4, A2        ; add offset to planes addr   *)
  214AH,0012H,       (* MOVE.L  A2, 18(A0)    ; store in windowTop field    *)
  214AH,0016H,       (* MOVE.L  A2, 22(A0)    ; store in curLine field      *)
  7400H,             (* MOVEQ   #0, D2                                      *)
  3428H,000CH,       (* MOVE.W  12(A0), D2    ; copy top (Y) to scratch     *)
  0D468H,0010H,      (* ADD.W   16(A0), D2    ; add height (Y) to top       *)
  2801H,             (* MOVE.L  D1, D4        ; copy modulo to scratch      *)
  0C8C2H,            (* MULU    D2, D4        ; mult by Y offset            *)
  0D868H,000AH,      (* ADD.W   10(A0), D4    ; add left (X offset)         *)

  2449H,             (* MOVE.L  A1, A2        ; planes addr to scratch in A2 *)
  0D5C4H,            (* ADDA.L  D4, A2        ; add offset to planes addr   *)
  214AH,001AH,       (* MOVE.L  A2, 26(A0)    ; store in lastLine field     *)

  4CDFH,061FH,       (* MOVEM.L (A7)+, D0-D4/A1-A2  ; pop these registers   *)
  4E75H );           (* RTS                   ; exit back to M2             *)
 END InitWarpInfo;


(* REGISTER USAGE:- GotoXY
 *
 * A0 - Pointer to an initialized WarpInfo structure (wInfo)
 *
 * D0 - New X position or character column (x)
 * D1 - New Y position or character row    (y)
 *)
PROCEDURE GotoXY(wInfo{8}: WarpInfoPtr; x{0}: CARDINAL; y{1}: CARDINAL);
 (* $E- no entry/exit code *)
 BEGIN INLINE(
  3140H,001EH,       (* MOVE.W  D0, 30(A0)    ; set new X in curX field     *)
  0C2E8H,0024H,      (* MULU    36(A0), D1    ; mult new Y by modulo        *)
  0D2A8H,0012H,      (* ADD.L   18(A0), D1    ; add windowTop offset        *)
  2141H,0016H,       (* MOVE.L  D1, 22(A0)    ; store in curLine field      *)
  4E75H );           (* RTS                   ; exit back to M2             *)
 END GotoXY;


(* REGISTER USAGE:- GetXY
 *
 * A0 - Pointer to an initialized WarpInfo structure (wInfo)
 *
 * D0 - Returns the current X position or character column
 * D1 - Returns the current Y position or character row
 *)
PROCEDURE GetXY(wInfo: WarpInfoPtr; VAR x, y: CARDINAL);
 (* normal entry/exit code *)
 BEGIN
  (* entry code:- *) (* MOVE.L  A4, -(A7)      ; push A4                     *)
                     (* MOVE.L  -xx(PC), A4    ; A4 points to global data    *)
                     (* MOVE.L  A5, -(A7)      ; push A5                     *)
                     (* MOVE.L  A7, A5         ; A5 points to top of stack   *)

  SETREG(8, wInfo);  (* MOVE.L  20(A5), A0     ; get wInfo off stack to A0   *)
  INLINE(
  2228H,0016H,       (* MOVE.L  22(A0), D1     ; get curLine into D1         *)
  92A8H,0012H,       (* SUB.L   18(A0), D1     ; subt windowTop  offset      *)
  82E8H,0024H,       (* DIVU    36(A0), D1     ; divide by modulo            *)
  7000H,             (* MOVEQ   #0, D0                                       *)
  3028H,001EH );     (* MOVE.W  30(A0), D0     ; get curX into D0            *)

  x:= REG(0);        (* MOVE.L  16(A5), A3     ; move VAR x to scratch A3    *)
                     (* MOVE.W  D0, (A3)       ; copy D0 into VAR x          *)
  y:= REG(1);        (* MOVE.L  12(A5), A3     ; move VAR y to scratch A3    *)
                     (* MOVE.W  D1, (A3)       ; copy D1 into VAR y          *)

  (* exit code:- *)  (* UNLK    A5             ; pop A5                      *)
                     (* MOVE.L  (A7)+, A4      ; pop A4                      *)
                     (* MOVE.L  (A7)+, A0      ; pop return address in A0    *)
                     (* LEA     12(A7), A7     ; deallocate parameters       *)
                     (* JMP     (A0)           ; exit back to M2             *)
 END GetXY;
```

```
(* REGISTER USAGE:- WarpText
 *
 * A0 - Pointer to an initialized WarpInfo structure (wInfo)
 * A1 - Address of string of characters to type  (text)
 * A2 - Pointer to TextFont structure, charData Address
 * A3 - Address of Current line.
 * A4 - Scratch
 * A5 - Scratch
 *
 * D0 - Number of characters to type (count)
 * D1 - Character to be emitted
 * D2 - LoChar
 * D3 - HiChar
 * D4 - Current X position
 * D5 - Scratch
 * D7 - Bitplane modulo; add to get to next raster line in bitplane.
 *)
PROCEDURE WarpText(wInfo{8}: WarpInfoPtr; text{9}: ADDRESS; count{0}: LONGCARD);
 (* $E- no entry/exit code *)
 BEGIN INLINE(
  48E7H,7D3CH,        (* MOVEM.L D1-D5/D7/A2-A5, -(A7) ; push these registers *)

  7800H,              (* MOVEQ   #0, D4                                       *)
  3828H,001EH,        (* MOVE.W  30(A0), D4    ; get curX into D4             *)
  7A00H,              (* MOVEQ   #0, D5                                       *)
  3A28H,0020H,        (* MOVE.W  32(A0), D5    ; get lastX into D5            *)
  2F05H,              (* MOVE.L  D5, -(A7)     ; push it on stack (sp_1)      *)

  3A28H,0024H,        (* MOVE.W  36(A0), D5    ; get modulo into D5           *)
  2F05H,              (* MOVE.L  D5, -(A7)     ; push it on stack (sp_2)      *)

  2468H,0000H,        (* MOVE.L  0(A0), A2     ; A2 is index into textFont    *)

  7400H,              (* MOVEQ   #0, D2                                       *)
  142AH,0020H,        (* MOVE.B  32(A2), D2    ; get loChar from font         *)

  7600H,              (* MOVEQ   #0, D3                                       *)
  162AH,0021H,        (* MOVE.B  33(A2), D3    ; get hiChar from font         *)

  3A2AH,0026H,        (* MOVE.W  38(A2), D5    ; get modulo from font         *)
  2F05H,              (* MOVE.L  D5, -(A7)     ; push it on stack (sp_3)      *)

  3A2AH,0014H,        (* MOVE.W  20(A2), D5    ; get ySize from font          *)
  2F05H,              (* MOVE.L  D5, -(A7)     ; push it on stack (sp_4)      *)

  246AH,0022H,        (* MOVE.L  34(A2), A2    ; A2 is index into charData    *)

  2868H,0012H,        (* MOVE.L  18(A0), A4    ; get windowTop addr to A4     *)
  2F0CH,              (* MOVE.L  A4, -(A7)     ; push it on stack (sp_5)      *)

  2868H,001AH,        (* MOVE.L  26(A0), A4    ; get lastLine addr to A4      *)
  2F0CH,              (* MOVE.L  A4, -(A7)     ; push it on stack (sp_6)      *)

  2668H,0016H,        (* MOVE.L  22(A0), A3    ; get curLine addr to A3       *)

  7E00H,              (* MOVEQ   #0, D7                                       *)
  3E28H,0022H,        (* MOVE.W  34(A0), D7    ; get bPMod to D7              *)
  5340H,              (* SUBQ.W  #1, D0        ; decrement count              *)
  7200H,              (* MOVEQ   #0, D1        ; clear D1                     *)
  5344H,              (* SUBQ.W  #1, D4        ; decrement curX               *)

  (*    DoNextChar:                                                            *)
  5244H,              (* ADDQ.W  #1, D4        ; increment curX               *)
  (*    DoNextSinAdd:                                                          *)
  1219H,              (* MOVE.B  (A1)+, D1     ; get char to write into D1    *)
  0C01H,0020H,        (* CMPI.B  #32, D1       ; is it a space?               *)
  6700H,002AH,        (* BEQ     * + 42        ; if eq    goto Blank:         *)
  0B202H,             (* CMP.B   D2, D1        ; compare with loChar          *)
  6D00H,0046H,        (* BLT     * + 70        ; if lt    goto BoffoChar:     *)
  0B203H,             (* CMP.B   D3, D1        ; compare with hiChar          *)
  6D00H,0036H,        (* BLT     * + 54        ; if lt    goto DoNoChar:      *)

  284AH,              (* MOVE.L  A2, A4        ; copy charData to A4          *)
  9282H,              (* SUB.L   D2, D1        ; subt loChar from char        *)
  0D9C1H,             (* ADDA.L  D1, A4        ; add char to charData         *)

  (*    DoChar:                                                                *)
  2A4BH,              (* MOVE.L  A3, A5        ; copy current line to A5      *)
  0DBC4H,             (* ADDA.L  D4, A5        ; add curX to current line     *)

  2A2FH,0008H,        (* MOVE.L  8(A7), D5     ; ySize (sp_4) is loop cnt     *)
  5385H,              (* SUBQ.L  #1, D5        ; decrement loop cnt           *)
  (*    MoveChar:                                                              *)
  1A94H,              (* MOVE.B  (A4), (A5)    ; move line char to planes     *)
  0D9EFH,000CH,       (* ADDA.L  12(A7), A4    ; add modulo (sp_3) to charData *)
  0DBC7H,             (* ADDA.L  D7, A5        ; add bPMod to bitplane        *)
  51CDH,0FFF6H,       (* DBF     D5, * - 10    ; decrement loop goto MoveChar: *)

  (*    Blank:                                                                 *)
  0B8AFH,0014H,       (* CMP.L   20(A7), D4    ; compare lastX (sp_1) and curX *)
  6D00H,0042H,        (* BLT     * + 66        ; if lt   goto GotoDoNext:     *)

  (*    DoLF:                                                                  *)
  7800H,              (* MOVEQ   #0, D4        ; curX is zero now             *)
  (*    DoCR:                                   ; current line is A3           *)
  0B7D7H,             (* CMPA.L  0(A7), A3     ; compare lastLine (sp_6) to A3 *)
  6700H,0042H,        (* BEQ     * + 66        ; if eq   goto GotoNSA:        *)

  0D7EFH,0010H,       (* ADDA.L  16(A7), A3    ; add modulo (sp_2) to next line *)
  6000H,003AH,        (* BRA     * + 58        ; goto GotoNSA:                *)

  (*    DoNoChar:    ; do an empty-box character (hex 80)                     *)
  284AH,              (* MOVE.L  A2, A4        ; copy charData to A4          *)
  0D9FCH,0000H,0060H, (* ADDA.L  #96, A4       ; add (char minus loChar) to A4 *)
  60C8H,              (* BRA     * - 56        ; goto DoChar:                 *)

  (*    BoffoChar:    ; may be LF, FF, CR or any control char                  *)
  0C81H,0000H,000AH,  (* CMPI.L  #10, D1       ; is it a linefeed?            *)
  67DEH,              (* BEQ     * - 34        ; if eq    goto DoLF:          *)
  0C81H,0000H,000CH,  (* CMPI.L  #12, D1       ; is it a formfeed?            *)
  6700H,000CH,        (* BEQ     * + 12        ; if eq    goto DoFF:          *)
  0C81H,0000H,000DH,  (* CMPI.L  #13, D1       ; is it a carriage return?     *)
  67CEH,              (* BEQ     * - 50        ; if eq    goto DoCR:          *)
  60DAH,              (* BRA     * - 38        ; else     goto DoNoChar:      *)

  (*    DoFF:                                                                  *)
  7800H,              (* MOVEQ   #0, D4        ; make curX zero now           *)
  266FH,0004H,        (* MOVE.L  4(A7), A3     ; point at windowTop  (sp_5)   *)
  6000H,000AH,        (* BRA     * + 10        ; goto GotoNSA:                *)

  (*    GotoDoNext:                             ; decrement count              *)
  51C8H,0FF82H,       (* DBF     D0, * - 126   ; and,     goto DoNextChar:    *)
  6000H,0006H,        (* BRA     * + 6         ; else done,   goto WindUp:    *)

  (*    GotoNSA:                                ; decrement count              *)
  51C8H,0FF7CH,       (* DBF     D0, * - 132   ; and,   goto DoNextSinAdd:    *)
  (*    WindUp:                                 ; else done,   ready to exit   *)
  508FH,              (* ADDQ.L  #8, A7        ; deallocate stack (sp_1 - 6)  *)
  508FH,              (* ADDQ.L  #8, A7        ; (6 longwords = 24 bytes)     *)
  508FH,              (* ADDQ.L  #8, A7                                       *)

  3144H,001EH,        (* MOVE.W  D4, 30(A0)    ; store update curX (X)        *)
  214BH,0016H,        (* MOVE.L  A3, 22(A0)    ; store update curLine (Y)     *)

  4CDFH,3CBEH,        (* MOVEM.L (A7)+, D1-D5/D7/A2-A5 ; pop these registers *)
  4E75H );            (* RTS                   ; exit back to M2              *)
 END WarpText;

END WarpText.imp
```

Programmer's Shareware

Programming Languages/Toolkits

- ☐ Assemblers and Structured Programming Tools (ALA101)
  2 Macro Assemblers and 2 Structure Preprocessors!
- ☐ Debuggers/System Monitors (ALA102)
  Great for fixing a bug in your program or just exploring!
- ☐ Assembly Language Examples, Toolkits, Utilities (ALA103)
  Examples and libraries for Assembly, with source!
- ☐ C Language Utilities (ALC101)
  Make utility, execution profiler, C tutorial and more!
- ☐ Optimizing C Language Compiler (ALC102)
  Full featured C compiler with SOURCE CODE!
- ☐ C Language Examples, Toolkits and Utilities (ALC103)
  Examples and libraries for C, with source!
- ☐ Artificial Intelligence Languages (ALM101)
  Object-oriented Lisp and a Prolog interpreter with source!
- ☐ Misc. Programming Languages (ALM102)
  LOGO graphics, ICON and a PostScript interpreter!
- ☐ Modula-2 Utilities Disk (ALM103)
  Asm & C to Modula-2 conversion, and a Modula-2 library!

Programmers Editors/Word Processing

- ☐ WordWright Word Processor (AWP101)
  Word processor with outliner, mailmerge and Spell check!
- ☐ Programming and Text Editors (AWP102)
  3 Great editors! Multiple windows, LARGE files, etc.

Utilities

- ☐ Archive & Compression Utilities (AUT101)
  Put twice as much information on your disks!
- ☐ Anti-Virus & Vaccine Utilities (AUT102)
  Protect your Amiga from Computer Viruses!
- ☐ Printer Utilites (AUT103)
  Eight utilities and Printer Drivers for over 40 printers!
- ☐ Great Utilities Disk #1 (AUT104)
  Disk recovery, disk optimizer, task handling, and more!
- ☐ Great Utilities Disk #2 (AUT105)
  Programmers WorkBench, Calculator, Shell and more!
- ☐ Diagnostics/Error Detection and Prevention (AUT106)
  Hardware test, Disk test, Guru error prevention!
- ☐ ARP - AmigaDOS Replacement Project (AUT107)
  Add power, speed and more commands to AmigaDOS!

Recreational/Personal/Business

- ☐ RIM DataBase Management System (ABP106/ABP107)
  Relational Information Management System. With source!
- ☐ AnalytiCalc Spreadsheet (ABP104/ABP105)
  Large and Powerfull with many extra features!
- ☐ Great Games Volume I (AGA102)
  StoneHenge, Missile Command, MazeMan and more!
- ☐ Great Games Volume II (AGA103)
  Gravity Wars, BattleShip, BlackJack, Clue and more!

TOLL FREE ORDERING: 800/874/9375
Canada, MI, & Info: 517/628/2943
Or Send Check or Money Order.
Visa and MasterCard accepted.

Computer Solutions
P.O. Box 354
Mason, MI 48854

1-9 disks -$5.50 per disk
10 or more - $5.25 per disk
Small Order (less $20.00) add $3.00
MI residents add 4% sales tax

MAIL ORDERS BE SURE TO INCLUDE COMPLETE ADDRESS, DAYTIME PHONE AND CHECK OR CREDIT CARD NUMBER!

The worlds fastest programmers* use the new

JForth
Professional 2.0

from
Delta Research
Box 1051, San Rafael, CA, 94915, (415) 485-6867
Call or write for information.

JForth is an interactive programming language that compiles directly to 68000 machine code. With JForth, you can compile and interactively test your code in seconds. Our "Clone" feature can generate Royalty Free Applications as small as 3K by automatically removing unused code. Execution speed is comparable to 'C'.

Language Features

- 32 bit Forth '83 standard plus '79 & FIG,
- Compile from normal files or BLOCKs,
- Hashing for fast compilation, • Local Variables,
- Source Level Debugger with breakpoints,
- FFP single and IEEE double Floating Point,
- Disassembler plus RPN and Motorola Assemblers
- Object Oriented Dialect, source code included,
- Many utilities: Sort, File I/O, dump structures....

Amiga Support

- Call any Amiga library routine by name,
- Complete Amiga structure support plus precompiled include files,
- Easy toolboxes for Graphics & Menus,
- IFF support plus simple animation system with Wipes, Fades, Blits, etc,
- Extensive tutorials and many examples for beginners and experts alike.

* Mike Haas and Phil Burk used JForth to win the Worlds Fastest Programmer Contest, held in Anahiem, California, Nov. 1988.

Ask for JForth at a store near you. Suggested retail price = **$179.95**

Creating Amiga Devices

A primer for home-made devices

by Steve Simpson

Steve Simpson is an ex-astronomer and now freelance consultant programmer of the Amiga and IBM PC. His current interests include systems, DOS and MIDI utilities development. He is currently working on an improved SideCar/BridgeBoard interface when not engaged in building Z80-based MIDI instruments.

The device concept lies at the heart of the Amiga's device-independent I/O architecture. Software devices offer a consistent interface between the Amiga system and the outside world. Some devices, such as trackdisk.device and console.device, are loaded into RAM from Kickstart at bootup time; others, such as printer.device and serial.device, are loaded in from the *devs:* directory when they are opened by an application.

This article is from a series of articles that resulted from a larger project. Reference is made to the similarity between devices and run-time libraries; the reader should refer to my article 'Creating Run-Time Libraries' (*Transactor for the Amiga,* Vol.1, Issue 4) for a thorough description of how an Amiga library is implemented. This article describes how an Amiga device can be created and added to the Amiga system.

What is an Amiga device?

Amiga devices are intended to provide a consistent interface as possible for I/O devices. To achieve this consistency, a software device must have a number of standard functions, structures and routines. A hardware device will also require unique device-specific routines; these must easily be accomodated in the device definition.

Structurally, a device is very similar in appearance to that of an Amiga run-time library. The *Device* structure is defined in the include file *exec/devices.h* as:

```
struct Device {
   struct Library dd_Library;
};
```

The *Library* structure is defined in the include file *exec/libraries.h.*

Like a library, a device consists of a table of jump vectors followed by the library node and library information, which in turn is followed by a data area. Trackdisk.device, for example, consists of:

```
FFDC (-36)    JMP AbortIO ;   offsets
FFE2 (-30)    JMP BeginIO
FFE8 (-24)    JMP ExtFunc
FFEE (-18)    JMP Expunge
FFF4 (-12)    JMP Close
FFFA (-6)     JMP Open

0000          Library Node
              Library flags, version, checksum,
                 open count
              Data Area
```

As with libraries, the order of the items in the jump table remains constant. The jump table builds downwards in memory from the library node, or library base address. The jump destinations are not constant – they reflect addresses at which AmigaDOS has loaded the device routines into RAM.

If register A6 contains a pointer to the device structure, then the instruction

```
JSR -30(A6)
```

would execute the function BeginIO(). The offset in assembler can be defined as a constant or declared as a global constant in the source file and resolved during linking. In the latter case, the above call could be changed to:

```
JSR _LVOBeginIO(A6)
```

Besides the routines Open(), Close(), Expunge() and ExtFunc(), which form the basic library definition, two additional routines must be present for a device: BeginIO() and AbortIO().

BeginIO() is the principal general-purpose I/O handler routine provided by all devices. This routine takes as input a pointer to an IORequest structure, which contains information relevant to the I/O call in question. The IORequest structure is found in the header file *exec/io.h*:

```
struct IORequest {
   struct Message io_Message;
   struct Device *io_Device;
   struct Unit *io_Unit;
   UWORD io_Command;
   UBYTE io_Flags;
   BYTE io_Error;
};
/*
 * io_Message contains message information associated
   with the IORequest structure. Used by the device to
   return I/O request upon completion. Used internally
   by devices for I/O request* queuing.
 * io_Device is a pointer to a Device structure for the
   device associated with this IORequest. This is set by
   the device when opened.
 * io_Unit is a pointer to a Unit structure for a unit
   of the device. This is set by the device when opened.
 * io_Command is the command requested., This is either
   a standard or device-specific command.
 * io_Flags is used for special request options and
   device state.
 * io_Error is the error number returned upon request
   completion.
 */
```

Commands can either be system standard commands or device-specific. Standard commands, such as **CMD_READ** and **CMD_WRITE**, are satisfied by all devices, if possible; device-specific commands would refer to a specific device implementation, such as allocating a sound channel for the audio device.

The standard commands:

- **CMD_CLEAR** clears all internal device buffers.
- **CMD_FLUSH** tells the device to abort all pending I/O requests in the device-unit request queue. All pending requests are returned with an error.
- **CMD_INVALID** tells the device that the calling task is sending an invalid command.
- **CMD_READ** reads a number of data bytes from the device internal buffers into the buffer defined in the calling program.
- **CMD_RESET** resets the device unit and resets the device internal routines. All pending commands are lost.
- **CMD_START** restarts execution of a device command that was stopped using CMD_STOP.
- **CMD_STOP** immediately stops the data processing being done. Requests are queued.
- **CMD_UPDATE** writes all device internal buffers out to the physical device unit. It ensures media are up to date and that no unwritten data remain in the internal buffers.
- **CMD_WRITE** writes a number of data bytes from the buffer defined in the calling program into the device internal buffers.

The standard commands are defined in the header file *exec/io.h*. Device-specific commands can be appended to this list.

AbortIO() takes a pointer to the IORequest structure as input. The routine attempts to cancel previous I/O requests. This is particularly useful for cancelling all queued requests to the device. (Note: IORequest refers to the above structure, while I/O request refers to the actual request of an I/O operation by a program.)

Additional routines can be made part of the device definition, and are private to the device. They are assigned jump-table entries at negative offsets beyond the device standard jump-table entries. (An example is the raw-key conversion routine RawKeyConvert() for the console device.) The offset constants for such routines can be derived from their position beyond the standard device routines using the macros **LININIT** and **LIBDEF**:

LIBINIT and **LIBDEF** are defined in *exec/libraries.i* as:

```
LIB_VECTSIZE    EQU   6
LIB_RESERVED    EQU   4
LIB_BASE        EQU   $FFFFFFFA
LIB_USERDEF1    EQU   LIB_BASE-(LIB_RESERVED*LIB_VECTSIZE)
LIB_NONSTD      EQU   LIB_USERDEF
LIBINIT         MACRO *
   IFC   '\1',''
COUNT_LIBSET    LIB_USERDEF
   ENDC
   IFNC  '\1',''
COUNT_LIBSET    \1
   ENDC
   ENDM

LIBDEFMACRO     *
   \1 EQUCOUNT_LIB
COUNT_LIBSET    COUNT_LIB-LIB_VECTSIZE
   ENDM
```

LIBINIT defines the standard library routine offsets for Open(), Close(), Expunge() and ExtFunc(). **LIBDEF** can then be used to define offsets for additional library function vectors:

```
LIBINIT
   LIBFUNC Dev_BeginIO
   LIBFUNC Dev_AbortIO
   LIBFUNC Dev_Func
... and so on
```

The routine Dev_Func() would then be jumped to by using

```
LINKLIB  Dev_Func,A6
```

where register A6 contains a pointer to the device structure.

Device management

When a device is opened, it may start one or more tasks. Each task has a task control block that is associated with the device, and each task runs concurrently with the other routines, using Exec's multitasking capability. Each task also has a message port, permitting queuing of I/O requests. An example of a de-

vice task is the task started for each unit of the trackdisk device, one unit for each disk drive present in the system.

The IORequest structure can be regarded as a means of passing messages to device tasks. When the requested I/O command in the message is completed, the device will Reply() the message to the calling program. In this context, BeginIO(), SendIO() and DoIO() are instances of the Amiga message passing mechanism. QuickIO(), however, is a special case. More on this later.

Device I/O control takes one of two forms: synchronous or asynchronous. Synchronous device I/O is the simpler mechanism. Control does not return to a program calling BeginIO(), for example, until the command in the associated IORequest message has been satisfied. The returned IORequest message will contain an error code or zero. The calling program waits until the data is available. Furthermore, a synchronous device may not have any task control block (sub-task) associated with it.

Asynchronous I/O relies on a device having associated with it at least one task and message port. The device is then capable of both asynchronous and synchronous device I/O, I/O request queuing depending on the device implementation. For asynchronous I/O, requests are made by the calling program. If the device maintains the tasks status flag, it sends the message immediately to the task. Otherwise, it queues the message at the task's message port. The device task then removes the request and deals with the IO requests as quickly as possible, on a first-in, first-out basis.

The calling program receives the reply when the request has been satisfied. The calling program can either wait for this reply (synchronous IO) or it can get on with something else in the meantime (asynchronous IO).

Device units

One or more Unit structures can be associated with each device. The Unit structure is defined in header file *exec/devices.h*, and looks like this:

```
struct Unit {
   struct MsgPort *unit_MsgPort;
   UBYTE unit_Flags;
   UBYTE unit_pad;
   UWORD unit_OpenCnt;
};

/*
 * unit_MsgPort is a pointer to the unit task's message port
 * unit_Flags are flags maintained for request queuing
   and task control
 * unit_OpenCnt is the number of times this task unit
   has been opened
*/
```

The Unit structure is initialized by the device for each unit opened, and a pointer to it is returned in the IORequest structure. The unit number is specified as a parameter in the OpenDevice() call. The unit number can refer to a physical unit or to a new implementation of a device task. For example, each disk drive is regarded as a unique unit by the trackdisk device. This means that the same code looks after all the disk units, but that each unit has its own task control block, message port and data areas.

Device IO

Only the audio, printer and timer system devices use the IORequest structure described earlier. The other devices use an extension to it: the IOStdReq structure. IOStdReq is defined in the header file *exec/io.h*:

```
struct IOStdReq {
   struct Message io_Message;
   struct Device *io_Device;
   struct Unit *io_Unit;
   UWORD io_Command;
   UBYTE io_Flags;
   BYTE io_Error;
   ULONG io_Actual;
   ULONG io_Length;
   APTR io_Data;
   ULONG io_Offset;
   ULONG io_Reserved1;
   ULONG io_Reserved2;
};

/*
 * io_Actual is the actual number of bytes transferred
   during the requested IO operation.
 * io_Length is the requested number of bytes to
   transfer during the IO operation.
 * io_Data is a pointer to a data buffer defined in the
   calling program.
 * io_Offset is the byte offset specification for byte
   offset structured devices (ie. the trackdisk device).
 * io_Reserved1 is reserved for structure expansion
 * io_Reserved2 is reserved for sturcture expansion
*/
```

A particular device implementation may require an extension to the IOStdReq or IORequest structures. For example, the trackdisk device uses a structure IOExtTD to keep a record of information pertinent to the trackdisk device:

```
struct IOExtTD {
   struct IoStdReq iotd_Req;
   ULONG iotd_Count;
   ULONG iotd_SecLabel;
};

/*
 * iotd_Req is the IOStdReq block defined above.
 * iotd_Count keeps track of disk changes.
 * iotd_SecLabel is a pointer to a 16-byte buffer for
   disk sector label information.
*/
```

The IOStdReq structure contains the IORequest as a substructure. The IOStdReq structure is used primarily by the device to return data to the calling program. A custom structure for a user device must incorporate either IORequest or IOStdReq as the first field in the structure. The other fields can be used to control the device.

Making I/O requests

I/O requests can be interfaced to devices through the following functions:

- BeginIO() initiates an I/O request. The request is either synchronous or asynchronous depending on the device implementation.
- DoIO() initiates an I/O request and waits for its completion. This is a synchronous form of I/O and does not return to the caller until completion.
- SendIO() initiates an IO request without waiting for completion. This is asynchronous I/O and control is returned to the caller even if the request is not completed.
- AbortIO() attempts to cancel a previous IO request.
- WaitIO() is used to wait for completion of an asynchronous I/O request. Control is not returned to the caller until the request has been completed.
- CheckIO() is used to check if an asynchronous IO request has completed.

DoIO(), SendIO(), WaitIO() and CheckIO() are Exec functions found in *exec.library*. BeginIO() is defined as part of the device definition. AbortIO() is defined both as part of device definition and as a support function in *exec.library*.

BeginIO() and AbortIO() refer to interface routines in the standard function link library module.

QuickIO()

QuickIO() provides a shortcut around queues at the device task message port. The setting of the IOF_QUICK flag in io_Flags tells the device that the requesting program wants the I/O done quickly. The device task will perform the I/O request immediately, if possible, and then send the result back to the calling program. QuickIO() tells the device that the calling program is not interested in getting a reply to its message. If a device has no associated task, the device will perform I/O as usual. The response of the device to the IOF_QUICK flag is, of course, dependent on the implementation of the device.

The SendIO() function sets the IOF_QUICK flag, returning to the calling program before the request has been satisfied. DoIO() clears this flag and then calls WaitIO() to wait for request completion. Both functions call the device's BeginIO() routine.

More on units

As described above, a device unit represents a pointer to a task control block (TCB), and thence to a task's message port. Depending on a device's implementation, it may have one or more units, each unit possessing an associated TCB. A device unit permits queuing of I/O requests. Examples of some system device units are: the keyboard device (one unit); the timer device (two units); and the audio device (four units). The console device has an unlimited number of units.

Ultimately, all requests, whether synchronous or asynchronous, access the unit via the device BeginIO() function. Once in BeginIO(), checks are made for a set QuickIO() flag, and whether the command is defined as requiring immediate attention by the device. If immediate or QuickIO(), the ExecuteIO() routine is executed, otherwise the request message is queued at the unit task's message port, before being passed to the ExecuteIO() routine via a call from the unit task.

The unit task, on removing each I/O request message from its message port, passes a pointer to the I/O request structure to the ExecuteIO() routine. A pointer to the device structure is also passed. The unit task then sets a busy flag, thus ensuring that further requests are queued until processing of the current request is completed. Once in ExecuteIO(), the address of the command function is recovered from the command table and a jump is made to it. The behaviour of the command functions is also implementation dependent.

Upon command completion, control passes to TerminateIO(), where unit flags are reset and the request is replied to the task that made the I/O request using a call to ReplyMsg().

Device routines

A device's routines fall into two main categories: functions and commands. Commands are those routines that are executed when an I/O request is made; functions are accessed like library functions, via an offset jump relative to the device's library base address.

In your assembler source code, functions and commands must be declared as external references (using **XREF** <function name>) in the main device source code. They are then incorporated into the device program during linking.

Commands have an address entry in the device's command table. The io_Command field of the I/O request structure indicates the command to be executed. These routines perform the actual I/O operations. Usually the routines receive their parameters via the registers. They set up the io_Actual and io_Error fields of the I/O request structure before returning. Command routines written in C would recover their arguments from the registers, storing them in local variables defined within the routines. Any returns from the routines would be made as usual, via the D0 register.

Device functions may set up some special condition critical to correct device operation. Alternatively, they may offer access to a special device-dependent feature, like Raw-KeyConvert() for the console device. Functions are addressed rather like library functions, and require a similar interfacing method.

In C programming, routines expect their arguments to be pushed onto the stack. We access a device function via an off-

set reference to the device library base address, passing the arguments to the function via the processor registers. You will need to write these interfaces, and then recover the arguments from the stack, storing them in local variables within the C functions. These interfaces can extract the device library base pointer from the device structure. See "Creating Run-Time Libraries" for details.

Constructing a device

The component parts of the user device are assembled or compiled with the option to omit the code startup module (*.begin* for Manx users, and *startup.obj* for Lattice users). We don't need this load module because we are in essence providing our own when the device is installed into the system.

Options to produce large code and large data are also used in the compilation and assembly stages. The device load module name should have the suffix *.device* appended for reasons of consistency.

The basic device structure given earlier may be extended: in the example device, we maintain pointers and flags here. In C, the example extended device structure looks like this:

```
struct MyDev {
   struct Device md_Device;
   struct ExecBase *md_ExecBase;
   ULONG  md_SegList;
   struct DosBase *md_DosBase;
   struct Unit *md_units[MD_NUMUNITS];
   UBYTE  md_Flags;
   UBYTE  md_pad;
};

/*
* md_Device is the device's device structure
* md_ExecBase is a pointer to exec.library library base
* md_DosBase is a pointer to dos.library library base
* md_SegList is a BCPL pointer to the device's segment list
* md_Flags flags maintained by the device
* md_units array of pointers to device units
*/
```

Example device

The new (1989) *ROM Kernel Reference Manual: Includes and AutoDocs* (hereafter referred to as 'the RKM') contains, in Section G, a sample device and a sample library. This code is correct and provides a starting point for the creation of custom devices and libraries.

However, the previous edition of the RKM (for AmigaDOS 1.2) has a number of errors in its code for an example skeleton device that render it difficult to use without many changes.

The *mydev.asm* listing provides the code for a functioning device. This device is supported by include modules *asmsupp.i* and *mydev_def.i*. The device code accompanying this article was developed using the layout of the RKM device code as a guide.

Many of the details concerning device implementation are identical to those for run-time library initialization. The reader is again referred to the the article "Creating Run-Time Libraries".

One important difference between devices and libraries is the function table funcTable in *mydev.asm*. It contains the long word absolute addresses of the device routines. Their order in this table is important; a user's own device functions must appear after the AbortIO entry. The table is terminated with -1L.

The example device can have up to four units associated with it. There are also examples of the command functions, plus two additional commands that don't do very much and are only provided as examples.

System interface

An Amiga device must have a minimum of six standard functions: Open(), Close(), Expunge(), ExtFunc(), BeginIO() and AbortIO(). References to OpenDevice(), CloseDevice() and RemDevice() are eventually translated into calls to Open(), Close() and Expunge() respectively. A reference to either BeginIO(), DoIO() or SendIO() is eventually translated into a call to BeginIO().

DoIO() and SendIO() are defined in the standard link library, whilst BeginIO() is an interface module with a jump instruction to the device's BeginIO() routine. Both DoIO() and SendIO() call BeginIO() after setting flags.

Installing the device

The listing *add_dev.c* illustrates how a user-defined Amiga device is added to the system. The device is loaded into memory using LoadSeg(), and a call to MakeLibrary() initializes the device's jump table, sets up the data areas and executes the user's device initialization routine. As with libraries, MakeLibrary()'s parameters include a pointer to the table of device function addresses, a pointer to a data initialization table, a pointer to the user device initialization routine, the size of the space required by the device in bytes, and a BCPL pointer to the segment list returned from LoadSeg(). The space required is the size of the device structure plus the user data area. The call to MakeLibrary() looks like this:

```
myDevice = MakeLibrary (funcInit, structInit,
                devInit, dataSize, segList);
```

where myDevice is a pointer to the MyDev structure defined earlier.

The call

```
AddDevice(myDevice);
```

enters the device entry into the list of devices maintained by Exec, making the device available to programs.

There is another way to install a user device into the system. Save the device into the DEVS: logical device directory (usual-

ly :devs on the system disk). A call to OpenDevice() causes the system to search the list of available devices; if it doesn't find it, it looks for the device module in the DEVS: directory. If found, the system loads the device and initializes it. For this to occur, the structure of the device must conform to that given in *mydev.asm*.

Using the device

There are a number of steps to follow before device routines, functions and commands can be referenced.

The device requires a reply message port to which to return requests to the calling task. Use the routine CreatePort() to create this reply port. This routine is defined in the function library accompanying your compiler.

The user device may use either the IORequest or the IOStdReq structures. Alternatively, a device dependent IO request structure may be used. A call to CreateStdIO() or CreateExtIO() sets up the standard structures and initializes the message node. (See *Amiga Programmer's Guide, Vol. 2* for details.)

Call OpenDevice() to provide access to the internal routines. A particular device implementation may allow multiple units.

Upon exiting the program using the device, don't forget to call CloseDevice() and to free all allocated memory. The IO request structure should also be freed, together with the task reply port. You can use DeleteStdIO(), DeleteExtIO() and DeletePort() to achieve this.

The program *main.c* illustrates how these initializations are made. It shows how commands are issued to the device using the I/O request, and how you can call device functions. The supporting interface module *devfunc.asm* details the steps needed to jump to a device function.

Register usage

Exec often puts pointers into particular registers when calling routines in the device. Register A6 carries a pointer to the device structure or library structure; A0 and A1 are pointers to data; and D0 and D1 are data registers. A0, A1, D0 and D1 can be regarded as scratch registers, and their contents are not retained during execution of device routines.

Summary

I/O requests to Amiga devices look very similar; this is the power behind the Amiga's device-independent I/O model. AmigaDOS devices can present a uniform interface to AmigaDOS for its communications with the outside world. An Amiga device can interface with almost any form of hardware device – speakers, timer, digitizing tablet, laser printer, plotter and so forth.

The programs accompanying this article were developed using the Manx Aztec C compiler/assembler v3.60a. They were debugged using the C source level debugger *sdb*, and the assembler debugger *db*.

References and Further Reading

ROM Kernal Reference Manual: Includes and AutoDocs. Addison Wesley. 1989

ROM Kernal Reference Manual: Libraries and Devices. Addison Wesley. 1987

The 'Kickstart' Guide to the Amiga. Ariadne Software

Mortimore, E., *Amiga Programmer's Guide, Vol.2*. Sybex.

Simpson, S.K., "Creating Run-Time Libraries", *Transactor for the Amiga*, Jan.1989, vol.1, issue 4., p.30

Williams, S., *Programming the 68000*. Sybex.

Listing 1: *mydev.asm*

```
;-----------------------------------------------------------------
;
; mydev.asm -- example device code
;
; Example of how a user system device is constructed.
; Adapted from ROM Kernal Manual (RKM) device.
;
; History:
;    88-05-17  created
;    89-05-02  final version
;
; assemble:   as -c -d -l -isys2:asm mydev
; link:       ln -o mydev.device mydev sys3:lib/cl.lib
;
; Note: uses large code and data model with no startup module
;       (.begin for Manx, startup.obj for Lattice)
;
;-----------------------------------------------------------------

 SECTION     section

 NOLIST
 INCLUDE "mydev_def.i"
 LIST


;-----------------------------------------------------------------
; external definitions and references
;-----------------------------------------------------------------
 XDEF     BeginIO
 XDEF     AbortIO
 XDEF     Close
 XDEF     DevFunc
 XDEF     DevInit
 XDEF     Expunge
 XDEF     ExtFunc
 XDEF     Open
 XDEF     myName

 XREF     _AbsExecBase

 XLIB     Alert
 XLIB     AllocMem
 XLIB     AllocSignal
 XLIB     CloseLibrary
 XLIB     CreateProc
 XLIB     FindTask
 XLIB     FreeMem
 XLIB     GetMsg
 XLIB     OpenLibrary
 XLIB     PutMsg
 XLIB     Remove
 XLIB     RemTask
 XLIB     ReplyMsg
 XLIB     SetTaskPri
 XLIB     Signal
 XLIB     WaitPort

 INT_ABLES
```

```
 DEVINIT
  DEVCMD   MYDEVCMD
  DEVCMD   MYDEV_END

;-----------------------------------------------------------------------
;The first executable location.  This returns an error in case someone
;tried to run the device as a program
;-----------------------------------------------------------------------

DevStart:
        moveq  #0,d0
        rts

;-----------------------------------------------------------------------
; various definitions and equates
;-----------------------------------------------------------------------
MYPRIORITY    EQU     0    ;library priority
VERSION       EQU     1    ;major version no.
REVISION      EQU     2    ;latest revision no.


;-----------------------------------------------------------------------
;A romtag structure.
;'exec' and 'ramlib' look for for this structure during loading operation
;to discover such as where to start running it, for example.
;In C this would be equivalent to the Resident structure.
;-----------------------------------------------------------------------
initDevDescrip:          ;structure RT,0
        dc.w    RTC_MATCHWORD       ;UWORD RT_MATCHWORD
        dc.l    initDevDescrip      ;APTR  RT_MATCHTAG
        dc.l    EndDevCode          ;APTR  RT_ENDSKIP
        dc.b    RTF_AUTOINIT        ;UBYTE RT_FLAGS
        dc.b    VERSION             ;UBYTE RT_VERSION
        dc.b    NT_DEVICE           ;UBYTE RT_TYPE
        dc.b    MYPRIORITY          ;UBYTE RT_PRI
        dc.l    myName              ;APTR  RT_NAME
        dc.l    DevIdString         ;APTR  RT_IDSTRING
        dc.l    DevInit             ;APRT  RT_INIT
                                    ;LABEL RT_SIZE

    ;my device name
myName        MYDEVNAME

;-----------------------------------------------------------------------
;device identifier tag used in supporting the device
;format is 'name version.revision (dd mon yyyy)',<cr>,<lf>,<null>
;-----------------------------------------------------------------------
DevIdString    dc.b    'mydev 1.2 (02 May 1989)',13,10,0

    ;name of dos library to open
dosName       DOSNAME

    ;force word alignmemt
        ds.w    0

;-----------------------------------------------------------------------
;These data are used by the loading program as parameters to MakeLibrary()
;-----------------------------------------------------------------------
DevInit:
        dc.l    MyDev_Sizeof        ;data space size
        dc.l    funcTable           ;pointer to function initialisers
        dc.l    dataTable           ;pointer to data initialisers
        dc.l    initRoutine         ;routine to run

;-----------------------------------------------------------------------
;table of addresses of mydev functions
;-----------------------------------------------------------------------
funcTable:
    ;standard system routines - these must always be included
        dc.l    Open
        dc.l    Close
        dc.l    Expunge
        dc.l    ExtFunc
    ;my device's definitions
        dc.l    BeginIO
        dc.l    AbortIO
    ;my device's function
        dc.l    DevFunc
    ;function table end marker
        dc.l    -1

;-----------------------------------------------------------------------
;this data table initialises the static data structures.  The format is
;specified in exec/InitStruc routine's desciption in RKM.
;The first argument is the offset from the device's library base for this
;byte/word/long.  The second argument is the value to be put into that
;cell.
;The table is NULL terminated
;-----------------------------------------------------------------------
dataTable:
        INITBYTE    LN_TYPE,NT_DEVICE
        INITLONG    LN_NAME,myName
        INITBYTE    LIB_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
        INITWORD    LIB_VERSION,VERSION
        INITWORD    LIB_REVISION,REVISION
        INITLONG    LIB_IDSTRING,DevIdString
        dc.l        0

;-----------------------------------------------------------------------
;This is the initialisation routine and is called after the device has
;been allocated.
;If it returns non-zero then the device will be linked into the device
;list maintained by Exec
;  registers:  d0-dev.ptr., a0-seg.list, a5-dev.ptr.
;-----------------------------------------------------------------------
initRoutine:
        move.l      a5,-(sp)
        move.l      d0,a5

        move.l      a6,md_ExecBase(a5)

        move.l      a0,md_SegList(a5)

        lea         dosName(pc),a1          ;open the dos.library
        moveq       #0,d0
        CALLSYS     OpenLibrary

        move.l      d0,md_DosBase(a5)       ;save the dos base
        bne.s       init_DosOK

        ALERT       AG_OpenLib!AO_DOSLib    ;can't open dos

init_DosOK:
    ;device specific initialisations go here
    ;
        move.l      a5,d0                   ;return ptr.to
        move.l      (sp)+,a5
        rts


;-----------------------------------------------------------------------
;System interface commands begin here.  Calling OpenDevice()/CloseDevice/
;RemDevice() translates into a call to the routines Open/Close/Expunge
;respectively.  The device library base pointer is in A6.  Task switching
;has been turned off while in these routines (via Forbid/Permit)
;so don't hang about.
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
;If there is an error the IO_ERROR field of the device request structure
;is set, otherwise a pointer to a device unit is setup in the IO_UNIT
;field.
;  registers: a6-dev.ptr. a1-ioreq.ptr. d0-unitnum d1-flags
;-----------------------------------------------------------------------
Open:
        movem.l     d2/a2-a4,-(sp)
        move.l      a1,a2               ;save the ptr. to the ioreq.struct.

        moveq       #MD_NUMUNITS,d2
        cmp.l       d2,d0               ;unit no.in range ?
        bcc.s       OpenError           ;unit number out of range

        move.l      d0,d2               ;save unit number
        lsl.l       #2,d0
        lea.l       md_Units(a6,d0.l),a4
        move.l      (a4),d0
        bne.s       OpenUnitOK          ;unit is already initialised

        bsr         InitUnit            ;try to get a unit

        move.l      (a4),d0
        beq.s       OpenError           ;error on opening

OpenUnitOK:
        move.l      d0,a3               ;unit pointer in a3

        move.l      d0,IO_UNIT(a2)
```

```
        addq.w      #1,LIB_OPENCNT(a6)     ;inc.device open counter
        addq.w      #1,UNIT_OPENCNT(a3)    ;inc.unit open counter

        bclr        #LIBB_DELEXP,md_Flags(a6)    ;disable delayed expunges

OpenEnd:
        movem.l     (sp)+,d2/a2-a4
        rts

OpenError:
        move.b      #IOERR_OPENFAIL,IO_ERROR(a2)
        bra.s       OpenEnd

;-----------------------------------------------------------------------
;If the device is no longer open, Close returns the segment list if the
;delayed expunge flag is set, otherwise the return is NULL
;  registers: a6-dev.ptr. a1-ioreq.ptr.
;-----------------------------------------------------------------------
Close:
        movem.l     a2-a3,-(sp)

        move.l      a1,a2

        move.l      IO_UNIT(a2),a3

        moveq.l     #-1,d0
        move.l      d0,IO_UNIT(a2)      ;reset unit ptr.
        move.l      d0,IO_DEVICE(a2)    ;reset device ptr.

        subq.w      #1,UNIT_OPENCNT(a3)    ;dec.unit open counter
        bne.s       CloseDevice            ;only if no units open

        bsr         ExpungeUnit

CloseDevice:
        subq.w      #1,LIB_OPENCNT(a6)     ;dec.device open counter
        bne.s       CloseEnd               ;only if device not in-use

        btst        #LIBB_DELEXP,md_Flags(a6)    ;any pending delayed expunges?
        beq.s       CloseEnd

        bsr         Expunge

CloseEnd:
        movem.l     (sp)+,a2-a3
        rts

;-----------------------------------------------------------------
;If the device is no longer open, then Expunge returns the segment
;list for the device code.  Otherwise the delayed expunge flag is set
;and NULL is returned in D0.
;Expunge may never wait or take a long time to complete.
;  registers: a6-dev.ptr.
;-----------------------------------------------------------------
Expunge:
        movem.l     d2/a5-a6,-(sp)
        move.l      a6,a5
        move.l      md_ExecBase(a5),a6

        tst.w       LIB_OPENCNT(a5)     ;is device open?
        beq         1$

        bset        #LIBB_DELEXP,md_Flags(a5)  ;set delayed expunge flag
        moveq       #0,d0
        bra.s       ExpungeEnd

1$:
        move.l      md_SegList(a5),d2   ;remove the device

        move.l      a5,a1
        CALLSYS     Remove              ;unlink from device list

    ;
    ;device specific closings go here
    ;

        move.l      md_DosBase(a5),a1   ;close the dos library
        CALLSYS     CloseLibrary

        moveq       #0,d0               ;free allocated memory
        move.l      a5,a1
        move.w      LIB_NEGSIZE(a5),d0
        sub.l       d0,a1
        add.w       LIB_POSSIZE(a5),d0
        CALLSYS     FreeMem

        move.l      d2,d0               ;return value

ExpungeEnd:
        movem.l     (sp)+,d2/a5-a6
        rts

;-----------------------------------------------------------------
;ExtFunc just returns 0 in D0 register
;-----------------------------------------------------------------
ExtFunc:
        moveq       #0,d0
        rts

;-----------------------------------------------------------------
;InitUnit initialises a device unit and starts up the associated
;unit task.
;  registers:  a6-dev.ptr.  a3-scratch  d2-unit no.
;-----------------------------------------------------------------
InitUnit:
        movem.l     d2-d4,-(sp)

        move.l      #MyDevUnit_Sizeof,d0      ;alloc.memory for unit struct
        move.l      #MEMF_PUBLIC!MEMF_CLEAR,d1
        LINKSYS     AllocMem,md_ExecBase(a6)

        tst.l       d0                        ;good allocation?
        beq         InitUnitEnd

        move.l      d0,a3
        move.b      d2,mdu_UnitNum(a3)        ;initialise unit number
        move.b      #NT_MSGPORT,LN_TYPE(a3)   ;init.MsgPort type

    ;
    ;We start the unit process here.
    ;
        move.l      #MYDEV_STACKSIZE,d4       ;set stack size
        move.l      #MyDevProcSeglist,d3      ;set segment list
        lsr.l       #2,d3                     ;change to BCPL pointer
        moveq       #MYDEV_PRIORITY,d2        ;set process priority
        move.l      #myName,d1                ;process name is device's
        LINKSYS     CreateProc,md_DosBase(a6) ;return MsgPort

        tst.l       d0                        ;process MsgPort
        beq         InitUnitFreeUnit

        move.l      d0,a0
        move.l      d0,mdu_Process(a3)        ;address of process MSgPort
        lea         -pr_MsgPort(a0),a0        ;address of taskTCB
        move.b      #PA_IGNORE,MP_FLAGS(a3)   ;flag=PA_IGNORE until reset
                                              ;by new process.
        move.l      a0,MP_SIGTASK(a3)

    ;here: a0-msg.port  a1-message
        lea         mdu_Msg(a3),a1
        move.l      a3,mdm_Unit(a1)           ;copy unit ptr.
        move.l      a6,mdm_Device(a1)         ;copy device ptr.
        move.b      #NT_MESSAGE,LN_TYPE(a1)   ;node type = message
        move.l      d0,a0                     ;message port is new process port
        LINKSYS     PutMsg,md_ExecBase(a6)    ;send startup msg.to process

        move.b      mdu_UnitNum(a3),d0        ;recover unit number
        lsl.l       #2,d0
        move.l      a3,md_Units(a6,d0.l)      ;save unit vector in unit table
                                              ;unit is now running

InitUnitEnd:
        movem.l     (sp)+,d2-d4
        rts

InitUnitFreeUnit:
        bsr         FreeUnit
        bra.s       InitUnitEnd

FreeUnit:
        move.l      a3,a1                     ;copy unit ptr.
        move.l      #MyDevUnit_Sizeof,d0
        LINKSYS     FreeMem,md_ExecBase(a6)   ;a6-dev.ptr.
        rts

ExpungeUnit:
        move.l      d2,-(sp)
```

```
;remove the unit task - at this point the unit open count is zero - so
;this is quite safe.
        move.l      mdu_Process(a3),a1
        lea         -(pr_MsgPort)(a1),a1
        LINKSYS     RemTask,md_ExecBase(a6)  ;a6-dev.ptr.

        moveq       #0,d2
        move.b      mdu_UnitNum(a3),d2       ;save unit no. (a3-unit ptr.)

        bsr         FreeUnit                 ;free unit struct.

        lsl.l       #2,d2
        clr.l       md_Units(a6,d2.l)        ;clear unit vector in device
                                             ;struct

        move.l      (sp)+,d2

        rts


;-----------------------------------------------------------------------
;The device specific commands begin here
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
;cmdTable - addresses of routines implementing commands are held here
;Address is aquired in the ExevrcuteIO call
;-----------------------------------------------------------------------
cmdTable:
        dc.l    Invalid         ;$00000001  bit no. 0
        dc.l    Reset           ;$00000002          1
        dc.l    Read            ;$00000004          2
        dc.l    Write           ;$00000008          3
        dc.l    Update          ;$00000010          4
        dc.l    Clear           ;$00000020          5
        dc.l    Stop            ;$00000040          6
        dc.l    Start           ;$00000080          7
        dc.l    Flush           ;$00000100          8
        dc.l    Custom          ;$00000200          9

;which commands are not to be queued
IMMEDIATE   EQU     $000001c2     ;Flush | Start | MyStop | MyReset

;-----------------------------------------------------------------------
;BeginIO - all incomimg IO enters through this routine.  The IO is either
;queued or is executed immediately
;  registers: a1-ioreq.ptr.  a6-dev.ptr.
;-----------------------------------------------------------------------
BeginIO:
        move.l      a3,-(sp)

        move.l      IO_UNIT(a1),a3          ;copy dev.unit vector

        moveq       #0,d0
        move.w      IO_COMMAND(a1),d0
        cmp.w       #MYDEV_END,d0           ;IO command in range?
        bcc.s       BeginIO_BadCmd

        DISABLE     a0

        btst        #IOB_QUICK,IO_FLAGS(a1)     ;quick flag set?
        bne.s       BeginIO_Immediate

        move.w      #IMMEDIATE,d1           ;is command immediate?
        btst        d0,d1
        bne.s       BeginIO_Immediate

        btst        #MDUB_STOPPED,UNIT_FLAGS(a3)   ;unit stopped?
        bne.s       BeginIO_QueueMsg               ;go Q message

        bset        #UNITB_ACTIVE,UNIT_FLAGS(a3)   ;is unit busy?
        beq.s       BeginIO_Immediate              ;no - do command

BeginIO_QueueMsg:
        bset        #UNITB_INTASK,UNIT_FLAGS(a3)   ;device needs attention
        bclr        #IOB_QUICK,IO_FLAGS(a1)        ;clear quick flag

        ENABLE      a0

        move.l      mdu_Msg(a3),a2          ;ptr.to message structure
        move.l      a1,mdm_Iob(a2)          ;ptr.to ioreq. struct.
        move.l      a2,a1
        move.l      mdu_Process(a3),a0      ;target process MsgPort
        LINKSYS     PutMsg,md_ExecBase(a6)  ;A0-target MsgPort, A1-Message

        bra.s       BeginIO_End

BeginIO_Immediate:
        ENABLE      a0
        bsr         ExecuteIO               ;do the IO

BeginIO_End:
        move.l      (sp)+,a3
        rts

BeginIO_BadCmd:
        move.b      #IOERR_NOCMD,IO_ERROR(a1)   ;set return error code
        bra.s       BeginIO_End

;-----------------------------------------------------------------------
;ExecuteIO does the actual dispatching of the request.  A3 has the unit
;pointer, A6 has the device pointer, A1 has the io request.
;  registers: a1-ioreq.ptr.  a3-unit ptr.  a6-dev.ptr.
;-----------------------------------------------------------------------
ExecuteIO:
        move.l      a2,-(sp)
        move.l      a1,a2               ;copy ioreq.struct.ptr.

        move.b      #0,IO_ERROR(a2)     ;reset error field

        moveq       #0,d0
        move.w      IO_COMMAND(a2),d0   ;copy command
        lsl.l       #2,d0               ;byte offset into table
        lea         cmdTable(pc),a0     ;index command address table
        move.l      0(a0,d0.w),a0

        jsr         (a0)                ;jump to command routine

        move.l      (sp)+,a2
        rts

;-----------------------------------------------------------------------
;TerminateIO sends the request back to the caller.  The device is not
;set to inactive if request was immediate or if request was started from
;the unit task
;  registers:  a1-ioreq.ptr.  a3-unit ptr.  a6-dev.ptr.
;-----------------------------------------------------------------------
TerminateIO:
        moveq       #0,d0
        move.w      IO_COMMAND(a1),d0       ;copy command
        move.w      #IMMEDIATE,d1
        btst        d0,d1                   ;is command immediate
        bne.s       TerminateIO_Immediate

        btst        #UNITB_INTASK,UNIT_FLAGS(a3)  ;unit not active now?
        bne.s       TerminateIO_Immediate

        bclr        #UNITB_ACTIVE,UNIT_FLAGS(a3)  ;turn off active bit

TerminateIO_Immediate:
        btst        #IOB_QUICK,IO_FLAGS(a1)     ;quick bit set?
        bne.s       TerminateIO_End

        LINKSYS     ReplyMsg,md_ExecBase(a6)    ;reply msg.to caller

TerminateIO_End:
        rts


;-----------------------------------------------------------------------
;AbortIO - this is not implemented here
;  registers:  a6-dev.ptr.  a1-ioreq.ptr.
;-----------------------------------------------------------------------
AbortIO:
        rts

AbortIO_End:
        rts


;-----------------------------------------------------------------------
;The functions that implement the device commands begin here.
;The following register usage is adopted:
;   A1 - ptr.to the io request block
;   A2 - another ptr.to the io request block
;   A3 - ptr.to the unit structure
;   A6 - ptr.to the device structure
;-----------------------------------------------------------------------
```

```
;-----------------------------------------------------------------------
;Invalid - returns just IOERR_NOCMD in io_error field
;-----------------------------------------------------------------------
Invalid:
        move.b      #IOERR_NOCMD,IO_ERROR(a1)  ;set return error code
        move.l      #0,IO_ACTUAL(a1)           ;no.bytes processed
        bsr         TerminateIO
        rts

;-----------------------------------------------------------------------
;Reset - this is not implemented
;-----------------------------------------------------------------------
Reset:
        move.l      #0,IO_ACTUAL(a1)      ;no.bytes processed
        rts

;-----------------------------------------------------------------------
;Read - acts as an infinite source of 1's.  Returns no. 1's in buffer
;-----------------------------------------------------------------------
Read:
        move.l      IO_DATA(a1),a0        ;ptr.to data buffer
        move.l      IO_LENGTH(a1),d0      ;no.bytes to process
        moveq       #0,d2                 ;no.bytes processed counter
        move        #0,ccr

        beq.s       ReadEnd               ;zero length?

        move.l      #1,d1                 ;value to copy

ReadLoop:
        move.b      d1,(a0)+
        add.l       #1,d2                 ;inc.bytes counter
        sub.l       #1,d0                 ;dec.no.reiterations
        bne.s       ReadLoop              ;if Z not set (d0!=0)

ReadEnd:
        move.l      d2,IO_ACTUAL(a1)      ;no.bytes processed
        bsr         TerminateIO
        rts

;-----------------------------------------------------------------------
;Write - dumps the contents of the request buffer.  Returns no.bytes
;written in io_actual field
;-----------------------------------------------------------------------
Write:
        move.l      IO_LENGTH(a1),IO_ACTUAL(a1)
        bsr         TerminateIO
        rts

;-----------------------------------------------------------------------
;Update - not implemented - should write out all pending io, not
;returning until done.
;-----------------------------------------------------------------------
Update:
        bra         Invalid

;-----------------------------------------------------------------------
;Clear - not implemented - should clear all internal buffers
;-----------------------------------------------------------------------
Clear:
        bra         Invalid

;-----------------------------------------------------------------------
;Stop - stops all io request from being processed until Start command
;is received
;-----------------------------------------------------------------------
Stop:
        bset        #MDUB_STOPPED,UNIT_FLAGS(a3)   ;set unit stopped bit
        move.l      #0,IO_ACTUAL(a1)
        bsr         TerminateIO
        rts

;-----------------------------------------------------------------------
;Start - starts a previously stopped unit task
;-----------------------------------------------------------------------
Start:
        bsr         InternalStart
        move.l      a2,a1
        move.l      #0,IO_ACTUAL(a1)
        bsr         TerminateIO
        rts

InternalStart:
        bclr        #MDUB_STOPPED,UNIT_FLAGS(a3)    ;reset unit stopped bit
        moveq       #0,d0
        moveq       #0,d1
        move.b      MP_SIGBIT(a3),d1          ;signal bit no.
        bset        d1,d0
        move.l      a3,a1
        LINKSYS     Signal,md_ExecBase(a6)    ;signal task to restart

        rts

;-----------------------------------------------------------------------
;Flush - returns all pending io requests to the caller.  It wouldn't
;be nice to kill processing in progress and let a request by.
;-----------------------------------------------------------------------
Flush:
        movem.l     d2/a6,-(sp)

        move.l      md_ExecBase(a6),a6

        bset        #MDUB_STOPPED,UNIT_FLAGS(a3)   ;task is stopped
        sne         d2

FlushLoop:
        move.l      a3,a0
        CALLSYS     GetMsg                         ;get request message

        tst.l       d0                             ;any messages?
        beq.s       FlushEnd                       ;no - jump

        move.l      d0,a1
        move.b      #IOERR_ABORTED,IO_ERROR(a1)    ;set io error code
        CALLSYS     ReplyMsg                       ;reply to caller

        bra.s       FlushLoop                      ;continue

FlushEnd:
        move.l      d2,d0
        movem.l     (sp)+,d2/a6

        tst.b       d0
        beq.s       1$

        bsr         InternalStart                  ;restart the process

1$:
        move.l      a2,a1
        bsr         TerminateIO

        rts

;-----------------------------------------------------------------------
;Custom - custom command - this doesn't do much
;-----------------------------------------------------------------------
Custom:
        bsr    TerminateIO
        rts


;-----------------------------------------------------------------------
;The device functions begin here
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
;DevFunc - returns the open count for a specific unit
;  registers:  a6-dev.ptr.  d0-unit no.
;-----------------------------------------------------------------------
DevFunc:
        move.l    a3,-(sp)

        move.l    #MD_NUMUNITS,d2
        cmp.l     d2,d0
        bcc.s     DevFuncError            ;unit no.in range?

        lsl.l     #2,d0
        move.l    md_Units(a6,d0.l),a3    ;recover unit ptr.
        cmp.l     #0,a3
        beq.s     DevFuncError            ;not a good ptr.

        move.w    UNIT_OPENCNT(a3),d0     ;copy unit open count

DevFuncExit:
        move.l    (sp)+,a3
        rts

DevFuncError:
```

```
        move.l    #-1,d0                ;flag error
        bra.s     DevFuncExit


;-----------------------------------------------------------------------
;The device unit process begins here
;
;A process is used so that queued requests may be processed at a later
;time.
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
;This bit cheats DOS - we fake having got this code into memory using
;LoadSeg.  Using LoadSeg the segment has the following prefix:
;  <seg.len><ptr.to next seg.>
;We set up this segment list here, and the process starts at the first
;address after the segment list
;-----------------------------------------------------------------------

MyDevProcStart:
 CNOP      0,4                    ;long word align (Manx feature only?)
        dc.l        16            ;segment length
MyDevProcSeglist:
        dc.l        0             ;ptr.to next segment

;-----------------------------------------------------------------------
;The process begins here
;-----------------------------------------------------------------------
MyDevProcBegin:
        move.l      _AbsExecBase,a6

        sub.l       a1,a1                 ;<my task> = FindTask(0)
        CALLSYS     FindTask

        move.l      d0,a0
        move.l      d0,a4                 ;save task in a4
        lea         pr_MsgPort(a0),a0     ;get msg port for my process
        move.l      a0,d4                 ;save task MsgPort
        CALLSYS     WaitPort              ;wait for startup msg.

        move.l      d0,a1
        move.l      d0,d2                 ;save the msg.ptr.
        CALLSYS     Remove                ;remove from msg.list

        move.l      d2,a2
        move.l      mdm_Device(a2),a5     ;ptr.to our new device in a5
        move.l      mdm_Unit(a2),a3       ;ptr.to our new unit in a3

        move.l      a5,d5                 ;save device ptr.
        move.l      a3,d3                 ;save unit ptr.

        moveq       #-1,d0                ;-1 is any signal at all
        CALLSYS     AllocSignal           ;allocate signal

        move.l      d3,a3                 ;recover unit ptr.
        move.b      d0,MP_SIGBIT(a3)
        move.b      #PA_SIGNAL,MP_FLAGS(a3)

;-----------------------------------------------------------------------
;The main loop starts here.
;procedure:
;  - wait for a mesage at the port
;  - lock the device - task is currently active and processing
;  - get a message - if no message then unlock the device
;  - call ExecuteIO and process the message
;  - loop back to wait and get a new message
;-----------------------------------------------------------------------
        bra.s       MyDevProcCheckStatus

MyDevProcMainLoop:
        move.l      d4,a0                 ;MsgPort for this process
        CALLSYS     WaitPort              ;wait for a msg.on the port

MyDevProcCheckStatus:
        move.l      d3,a3
        btst        #MDUB_STOPPED,UNIT_FLAGS(a3) ;is device stopped?
        bne.s       MyDevProcMainLoop            ;yes

        bset        #UNITB_ACTIVE,UNIT_FLAGS(a3)  ;set process flag - Q msg.
        beq.s       MyDevProcMainLoop         ;device in use when Z not set

MyDevProcNextMessage:
        move.l      d4,a0                 ;process MsgPort
        CALLSYS     GetMsg                ;get next request msg.
        tst.l       d0
        beq.s       MyDevProcUnlock       ;no message ?

        move.l      d0,a2                 ;copy msg.ptr.
        move.l      d3,a3                 ;MyDevUnit ptr.
        move.l      mdm_Iob(a2),a1        ;ioreq.block ptr.
        exg         a5,a6                 ;before-A6=ExecBase
        bsr         ExecuteIO             ;handle the msg.
        exg         a5,a6                 ;reset ExecBase address

        bra.s       MyDevProcNextMessage

MyDevProcUnlock:
        and.b       #$ff&(UNITB_ACTIVE!UNITB_INTASK),UNIT_FLAGS(a3)
        bra         MyDevProcMainLoop

MyDevProcFail:
        bsr         FreeUnit              ;here when initialisation fails
        rts

;-----------------------------------------------------------------------
;This is the end of our device code
;-----------------------------------------------------------------------
EndDevCode:
         END
```

Listing 2: *asmsupp.i*

```
;-----------------------------------------------------------------------
; asmsupp.i
;
; supplemental macros
;
; History:
;    88-05-06  created
;-----------------------------------------------------------------------

LINKSYS          MACRO
    LINKLIB  _LVO\1,\2
    ENDM

CALLSYS          MACRO
    CALLLIB  _LVO\1
    ENDM

XLIB             MACRO
    XREF     _LVO\1
    ENDM
```

Listing 3: *mydev_dev.i*

```
;-----------------------------------------------------------------
; mydev_def.i
;
; assembler defines for mydev.device
;
; History:
;    88-05-16  created
;-----------------------------------------------------------------

 INCLUDE "exec/types.i"
 INCLUDE "exec/ables.i"
 INCLUDE "exec/alerts.i"
 INCLUDE "exec/devices.i"
 INCLUDE "exec/errors.i"
 INCLUDE "exec/initializers.i"
 INCLUDE "exec/io.i"
 INCLUDE "exec/libraries.i"
 INCLUDE "exec/lists.i"
 INCLUDE "exec/memory.i"
 INCLUDE "exec/nodes.i"
 INCLUDE "exec/resident.i"
 INCLUDE "libraries/dos.i"
 INCLUDE "libraries/dosextens.i"
 INCLUDE "asmsupp.i"


;device function offsets from base structure
 LIBINIT
  LIBDEF    Dev_BeginIO
  LIBDEF    Dev_AbortIO
```

```
 LIBDEF     Dev_Func

MD_NUMUNITS   EQU   4

;mydev device data area structure
 STRUCTURE   MyDev,LIB_SIZE
   ULONG  md_ExecBase
   ULONG  md_SegList
   ULONG  md_DosBase
   STRUCT md_Units,MD_NUMUNITS*4
   UBYTE  md_Flags
   UBYTE  md_pad
   LABEL  MyDev_Sizeof

 STRUCTURE MyDevMsg,MN_SIZE
   APTR   mdm_Device
   APTR   mdm_Unit
   APTR   mdm_Iob
   LABEL  MyDevMsg_Sizeof

 STRUCTURE MyDevUnit,UNIT_SIZE
   UBYTE  mdu_UnitNum
   UBYTE  mdu_pad
   APTR   mdu_Process
   STRUCT mdu_Msg,MyDevMsg_Sizeof
   LABEL  MyDevUnit_Sizeof

;status bit flag for unit stopped
 BITDEF  MDU,STOPPED,2

;stack size and priority for the process we will create
MYDEV_STACKSIZE  EQU    $500
MYDEV_PRIORITY   EQU    0

;the name of my device
MYDEVNAME          MACRO
      dc.b     'mydev.device',0
      ENDM
```

Listing 4: *devfunc.asm*

```
;-----------------------------------------------------------------
; devfunc.asm
;
; interface routine between C program and function in mydev.device
;
; History:
;   88-09-11  created
;
; assemble:  as devfunc
;-----------------------------------------------------------------

 NOLIST
 INCLUDE "mydev_def.i"
 LIST

 XREF  _DevFunc

;-----------------------------------------------------------------
; C calling sequence:  no_open = DevFunc(req,unit)
; registers:             D0              A0   D0
;
;                      LONG no_open, unit;
;                      struct IOStdReq *req;
;
;-----------------------------------------------------------------
_DevFunc:
    ;recover call data from stack
        move.l      4(sp),a0
        move.l      8(sp),d0

    ;call the device function
        LINKLIB  Dev_Func,IO_DEVICE(a0)

    ;return to caller
        rts

        end
```

...continued overleaf

Free Spirit Software Inc.

AMI..ALIGNMENT SYSTEM

Precision drive evaluation and head alignment system for the Amiga series of computers. The Ami..Alignment System evaluates the mechanical condition of internal and external 3.5" disk drives and provides information for the correction of non-standard conditions.

From a control panel menu, head alignment, drive speed and read/write performance may be tested. Instantaneous readouts are displayed on screen or may be output to printer.

Includes program disk, calibration disk and operations manual. Instructions and diagrams for dissassembly and adjustment of Commodore and third party drives are included.

Diagnose and treat the "flaky floppy" syndrome.

Ami...Alignment

See your local dealer or contact:
Free Spirit Software
58 Noble St.
Kutztown, PA 19530
215-683-5609
FAX #215-683-8567

Treat your drives with respect!

Suggested retail price $49.95

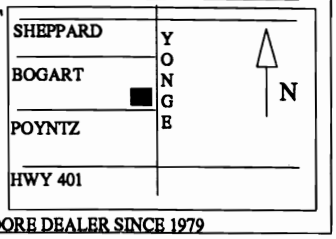
CANADA'S BEST SELECTION OF AMIGA - C64 - C128 SOFTWARE

COMSPEC HARD DRIVES AND BERNOULLI DRIVES AVAILABLE FOR THE AMIGA 500 AND 2000

Electronics 2001

COMPUTER PRODUCT CENTRE

4736 YONGE STREET
WILLOWDALE ,ONT
M2N 5M6
TEL: (416) 229-2700



AUTHORIZED COMMODORE DEALER SINCE 1979

New Store!

Fisher's
Computers & Software

Authorized Amiga Dealer

8005 Archibald
Rancho Cucamonga, CA 91730
(714) 987-1662

Serving The Inland Impire

Listing 5: *add_dev.c*

```
/*****************************************************************
 * add_dev.c
 *
 * Module to add a user device to the device list maintained
 * by the system.  After adding the user device is available to
 * all subsequent programs
 *
 * History:
 *   88-05-06  created
 *
 * compile:  cc add_dev
 * link:     ln add_dev -lc
 *****************************************************************/

#include <functions.h>
#include <exec/types.h>
#include <exec/libraries.h>
#include <exec/resident.h>
#include <exec/devices.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>


ULONG segList,      /* BCPL seglist pointer */
      codeloc;      /* pointer to code location */

struct Init {
  ULONG space;        /* data space size */
  ULONG funcTable;    /* ptr.to func offsets table */
  ULONG dataTable;    /* ptr.to initStruct table */
  ULONG initRoutine;  /* ptr.to library initRoutine */
} *init;

struct Resident *devRes;  /* ptr.to device rom tag area */

ULONG space, funcTable, dataTable, initRoutine;

struct Device *mydev;


main()
{

      /* load the device segment */
  segList = (ULONG)LoadSeg("mydev.device");
  if (segList==NULL) {
    printf("Can't load mydev.device\n");
    exit(0);
  }

      /* address of start of device code in memory
       * multiply by 4 since segList is a BCPL pointer */
  codeloc = segList * 4;

      /* address of device resident rom tag data area */
  devRes = (struct Resident *)(codeloc + 8);

      /* address to initialisation data table */
  init = (struct Init *)devRes->rt_Init;

      /* extract data from initialisation table */
  space = init->space;
  funcTable = init->funcTable;
  dataTable = init->dataTable;
  initRoutine = init->initRoutine;

      /* initialise the device */
  mydev = (struct Device *)MakeLibrary(funcTable, dataTable,
                                       initRoutine, space, segList);

      /* make the device known to the system */
  AddDevice(mydev);

  printf("mydev.device added to system\n");

}
```

Listing 6: *main.c*

```
/*****************************************************************
 * main.c
 *
 * Test program for mydev.device
 *
 * History:
 *   88-09-11  created
 *
 * compile:  cc main
 * link:     ln main devfunc -lc
 *****************************************************************/

#include <functions.h>
#include <exec/types.h>
#include <exec/libraries.h>
#include <exec/devices.h>
#include <exec/io.h>

#define CMD_CUSTOM  (CMD_FLUSH+1)

struct MsgPort *reply;
struct IOStdReq *req;
UBYTE dev_flag=0;
UBYTE buff[10];

main()
{
  ULONG ret, unit, val, no_open, DevFunc();
  WORD j;

    /* make reply port */
  reply = CreatePort("my_main_reply",0);
  if (reply==NULL) {
    printf("Can't create reply port\n");
    cleanup();
  }

    /* make io request */
  req = (struct IOStdReq *)CreateStdIO(reply);
  if (req==NULL) {
    printf("Can't create IO request\n");
    cleanup();
  }

    /* open the test device */
  ret = OpenDevice("mydev.device",0L,req,0L);
  if (ret!=NULL) {
    printf("Can't open mydev.device\n");
    cleanup();
  }
  dev_flag = 1;

    /* read something */
  req->io_Command = CMD_READ;
  req->io_Data = (APTR)buff;
  req->io_Length = (ULONG)sizeof(buff);
  DoIO(req);
  printf("io_Actual=%ld\n",req->io_Actual);

    /* try out our custom command */
  req->io_Command = CMD_CUSTOM;
  BeginIO(req);
  printf("io_Actual=%ld\n",req->io_Actual);

    /* call our custom device function */
  unit = 0L;
  no_open = DevFunc(req,unit);
  printf("no_open=%ld\n",no_open);

  cleanup();
}

/*****************************************************************
 * cleanup() - cleanup everything we had open/allocated
 *****************************************************************/
cleanup()
{

  if (dev_flag) {
    CloseDevice(req);
  }
  if (req) {
    DeleteStdIO(req);
  }
  if (reply) {
    DeletePort(reply);
  }
}
```

HOURS:
8 AM to 7 PM Monday - Friday
8 AM to 4 PM Saturday
Call on Sunday

The Software Shop, Inc.

Your AMIGA Software and Hardware Marketplace

Visa and MasterCard Accepted

Hardware Spectacular! Sensational Savings! Compare!

HardCards w/HF2000		HardCards w/FastCard		FastTrack A500/A1000		ToolBox A500/A1000		Memory Expansion A2000/A2500/A500/A1000			
HardFrame	$ 249	FastCard	$ 159	FastTrack Sys.	$ 379	ToolBox	$ 249	8Up 2Mg	$ 489	Starboard Ok	$ 269
138N 32Mg	589	138N 32Mg	506	138N 32Mg	722	ToolBox w/48Mg	849	8Up 4Mg	769	Starboard 512K	359
157N 48Mg	669	157N 48Mg	589	157N 48Mg	799	ToolBox w/32Mg	769	8Up 6Mg	1,059	Starboard 1Mg	449
Quantum 40Mg	759	Quantum 40Mg	689	Quantum 40Mg	899	ToolBox w/Q40	939	8Up 8Mg	1,349	Starboard 2Mg	669
Quantum 80Mg	1,089	Quantum 80Mg	1,039	Quantum 80Mg	1,223	ToolBox w/Q80	1,279	Spirit 512K	319	MiniMeg 512K	378
Quantum 105Mg	1,179	Quantum 105Mg	1,099	Quantum 105Mg	1,304	ToolBox w/Q105	1,349	Spirit 1Mg	411	MiniMeg 1Mg	499
277N 63Mg	759	277N 63Mg	689	277N 63Mg	889	ToolBox w/63Mg	939	Spirit 105Mg	499	MiniMeg 2Mg	735

External 5.25 Drv. $179 External 3.5 Drv. $119 Internal 3.5 Drv. $89 Digi-View Gold, WV1410 Panasonic camera w/lens and Copy stand $399

ACCESSORIES

A-B Switch cable	$35.00
A/B/C/D Switchbox	63.00
Boing optical mouse	109.00
Copy Stand	65.00
Ece Midi 1000	52.00
Ece Midi 500/2000	52.00
Epyx 500JX	15.00
Ergo Joystick	17.00
Freedom stick	65.00
Infared mouse	176.00
MD120 3.5 Disk holder	29.00
MD60 disk holder	26.00
MD64 disk holder	26.00
Modem cable A1000	15.00
Modem cable A2000/500	15.00
Mouse holder	6.00
Mouse master	29.00
Mouse pad	9.00
Multisync cable I-102M/F	24.00
Okimate black ribbon	5.00
Okimate color ribbon	8.00
Power strip 6 outlets	17.00
Printer cable 2000/500	15.00
Printer cable A1000	15.00
Quick shot Joystic	15.00
Ribbon panasonic all	10.00
Slick stick	8.00
Star NX black ribbon	8.00
Star NX color ribbon	12.00
Tilt and swivel stand	21.00
Xerox 4020 500/sheet	25.00

CAD SYSTEM

Draw 2000	$183.00
Home builder cad	129.00
Home builder choice	53.00
Intro cad	51.00
Logic works	65.00
Pro-Board	399.00
Pro-Net	399.00
X-Cad system	385.00

A500/A2000/A2500 PERIPHERALS

A1084S stereo monitor	CALL
A2000 HD 40mg28ms w/cont	CALL
A2058 2mg 8mg ram board	CALL
A20620/2meg 32bit	CALL
A20620/4mg 32bit Mem	CALL
A2090A HD controller	CALL
A2300 Genlock	CALL
A2500 HD40mg28ms3mgMem 68020/68881	CALL
Amiga 2000 Cpu	CALL
Amiga 500 CPU	CALL
AT Bridge card A2286	CALL
Bridge card A2000	CALL

COMMUNICATIONS

Atalk III	$65.00
Atredes profess.BBS	96.00
BBS pc	96.00
Diga	50.00
Online Platinum	69.00

CREATIVE PRODUCTIVITY

Analytic art	$39.00
Gizmoz enhanced	45.00
Intellitype	35.00
Magellan	129.00
Master type	25.00
Mavis beacon typing	33.00
Your family tree	32.00

DATABASE MANAGEMENT

Acquisition	$189.00
Data retrieve	51.00
Data Retrieve Prof	189.00
Dbman	125.00
Micro base	24.00
Micro fiche filer	64.00
Microfiche filer plus	115.00
Organize	49.00
Softwood file II gs	79.00
Super base	52.00
Superbase pers. II	98.00
Superbase Pro 3.0	229.00
Superbase professional	189.00

PROGRAMMING LANGUAGES

A/C basic	$129.00
A/C fortran	195.00
Algebra ii	33.00
Arexx	33.00
Assem pro	65.00
Aztec C developer	195.00
Aztec C professional	129.00
Aztic complete devel.sys.	315.00
Benchmark C library	60.00
Benchmark iff library	62.00
Benchmark modula 2	128.00
Benchmark simplified	62.00
Cape 68k	59.00
Critic's Choice	159.00
Devpac Ed/Assem/link/debg	66.00
Dissasembler	45.00
GFA basic 3.0	95.00
Inovatools #1	54.00
Lattice C	132.00
Lattice C + +	379.00
Lattice C professional	249.00
Lattice developemnt system 5.0	199.00
Library's source	197.00
Macro assembler	65.00
Metacomco lisp	131.00
Metacomco pascal	95.00
Metacomco shell	40.00
Metascope debugger	65.00
Power windows v2.5	58.00
Pro Basic	124.00
Publishers choice	130.00
Source level debugger	50.00
The power system pecan	93.00
The professional pecan	89.00
The Works Platinum	164.00
True basic	65.00
W. shell	33.00

FONTS

Asha's fonts	57.00
Calligrapher	$63.00
Fancy 3d fonts	52.00
Font set I	22.00
Inter font	76.00
Kara fonts color	50.00
Lion fonts	56.00
Masterpiece fonts	159.00
News letter fonts	29.00
Profonts I Prof	23.00
Profonts II decorative	23.00
Studio font	29.00

HARDWARES

8Up Memory Exp. A2000	$165.00
Accelerator	152.00
Amigen	153.00
Easyl A1000	349.00
Easyl A2000	379.00
Easyl A500	345.00
Expan. 512K A500	159.00
Expansion 1mg A500	409.00
Flicker fixer	495.00
Framegrabber	579.00
Future sound	129.00
Light pen	99.00
Live A2000	349.00
Live framegrabber A1000	245.00
Live framegrabber A500	305.00
MC68881 Math Co-processor	179.00
Multifunction card	75.00
My-T-Mouse	39.00
Perfect sound A1000	69.00
Perfect sound A500/A2000	69.00
Phoenix power supply A500	95.00
Progen	379.00
Super Gen	695.00
Supra modem 2400bd	149.00
WV1410 Panasonic Camera w/lens	199.00

SPREADSHEETS

Analyze! 2.0	$95.00
Haicalc	32.00
Maxiplan 500	95.00
Maxiplan plus	126.00
Super plan	97.00
VIP Professional	65.00

DESKTOP PUBLISHING

Citydesk	$97.00
Citydesk companion	15.00
Citydesk companion II	15.00
Page stream	129.00
Page setter	97.00
Professional Draw	129.00
Professional page	249.00
Publisher plus	69.00
Shakespeare	129.00

ACCOUNTING

The Accountant	$183.00
Best general ledger	77.00
Financial plus	189.00
Invester advantage	63.00
Miamiga ledger	63.00
Micro lawyer	39.00
Easy Ledger	199.00

WORDPROCESSORS

Becker text	$92.00
Excellence!	179.00
Flow idea processors	63.00
Gold spell II	30.00
Kindwords	65.00
LPD writer	75.00
LPD writer unprotected	83.00
Micro text	22.00
Prowrite v2.0	80.00
Reason	249.00
Text pro	50.00
Word perfect	159.00
Wp library	79.00

MONITORS

Amdek multisync 12' mon.	$512.00
Panasonic 8cm515 mon.	249.00
Sieko cm1430 monitor	699.00
Sony multi-scan Monitor	535.00
Zenith 14' Flat screen mon.	720.00
Nec 3D multisync	799.00
Taxan 1000 20" multisync	2899.00

EDUCATION

Adventure of sinbad	32.00
Aesop's fables	32.00
All about America	35.00
Animal kingdom	32.00
Decimal dungeon	32.00
Discovery game math	25.00
Discovery game spell	25.00
First letters and words	35.00
First shapes	35.00
Fraction action	32.00
Grade manager	59.00
Great states	25.00
Kid Talk	35.00
Kinderama	32.00
Kwik speak spanish	19.00
Learning curve	52.00
Link word French	22.00
Link word German	22.00
Link word Italian	22.00
Link word Russian	22.00
Link word Spanish	22.00
Math talk	35.00
Math talk fraction	35.00
Math wizard	35.00
Perfect score	51.00
Quiz master	51.00
Read & rhyme	32.00
Read-a-rama	32.00
Spell bound	25.00
Speller bee	35.00
Tales from Arabia	32.00
Wordmaster	32.00

PRINTERS

1091i Panasonic printer	$224.00
Alps Allegro 24 PIN	399.00
Epson Apex 80	205.00
NEC LC890 laser PS.	3395.00
Okimate 20 color prt.	145.00
Plug'n print	85.00
Star nx rainbow	259.00
Xerox 4020 ink jet	1295.00
Xerox 4020 starter kit	159.00
HP Paint jet	CALL

1-800-752-0050

In Mass. Call....

1-508-756-6452

SALES AND QUOTES (outside Mass.):
1-800-752-0050

SHIPPING AND TECHNICAL SERVICE:
1-508-756-6452

QUOTE-A-FAX: 1-508-799-9354

- Corporate and volume purchases invited. Call Moe.
- Stock orders placed by 12:00 NOON EST shipped the same day.
- Allow 10 days for personal checks to clear.
- Wire transfer accepted.
- Minimum $5.00 Shipping/handling charge
- Add $3.50 for COD.
- Returns subject to restocking charge
- Software defects-call the Manufacturer
- For faster delivery send cashiers check

SOUND & MUSICS

4-op deluxe	$97.00
Audio master	37.00
Audio master II	67.00
Back songbook	27.00
Copyist II	179.00
D-50	99.00
Deluxe music	69.00
Dr drums	20.00
Dr keys	20.00
Dr T's Keyboard	160.00
Dr. T's Midi recording studio	45.00
Dx heaver	97.00
Dynamic drums	52.00
Dynamic studio	129.00
Future sound A500/A2000	92.00
Hot cool jazz	18.00
Hot licks	32.00
Instant music	32.00
Matrix 6	97.00
Midi Magic	99.00
Mt-32	97.00
Music student	38.00
Pro sound designer	125.00
Promidi studio	129.00
Rock & roll	21.00
Sonix	51.00
Sound sampler	78.00
Studio magic	65.00
Synthia	59.00
Texture	97.00
Utilities 2(mimatics)	43.00

UTILITIES

B.A.D.disk optimizer	$32.00
C.B. Tree	65.00
Celebrity cook book	20.00
Custom screens *	45.00
Cygnused Professional	65.00
Disk 2 disk	33.00
Disk master	33.00
Disk mechanic	59.00
Dos 1.3	22.00
Dos 2 dos	35.00
Doug's math aquarium	52.00
Encore	45.00
Ez backup	33.00
Facc II	22.00
Fine print	33.00
Fleet check	33.00
Flip side	37.00
Gold spell	32.00
Gomf	27.00
Grabbit	19.00
Laser scripts	28.00
Laser up: fonts	26.00
Laser up: plot	33.00
Laser up: print	58.00
Laser up: utilities	26.00
Lex check	33.00
Lv backup	32.00
Multi preferences	20.00
Nag Plus Schedule Assist.	52.00
Pro script	32.00
Project D	32.00
Promis spelling	32.00
Quarterback	45.00
Raw copy 1.3	39.00
Superback	52.00
Sys font *	22.00
Tax plan	22.00
Tdi editor sources	30.00
Tdi examples	15.00
Tdi grid access	30.00
Tdi kermit	19.00
Text ed plus	52.00
V.I.P.	32.00

HOME ACCOUNTING

Desktop budget	$46.00
Money mentor	$65.00
Phaser	65.00
Tax Break	53.00

GRAPHICS & VIDEO

3-Demon	$69.00
Animate 3-D	99.00
Animate 4D	439.00
Animation editor	39.00
Animation effects	32.00
Animation Flipper	32.00
Animation multiplane	58.00
Animation stand	32.00
Animation w/images	89.00
Animator apprentices	185.00
Animotion	65.00
Architechtural design	23.00
Art gallery I	20.00
Art gallery II	20.00
Art pak I	20.00
Art parts 1	20.00
Art parts II	20.00
Award maker plus	32.00
Broadcast Titler	189.00
Butcher	25.00
C light 3d editor	39.00
Calligrapher help	28.00
Chroma Paint	48.00
Color splitters	79.00
Comicsetter	65.00
Deluxe paint II	89.00
Deluxe Paint III	102.00
Deluxe print II	55.00
Deluxe photo lab	102.00
Deluxe production	137.00
Deluxe video II	89.00
Designasaurus	32.00
Digi paint	39.00
Digi paint help	24.00
Digi view	131.00
Director's Toolkit	26.00
Dpaint help	24.00
Exprss paint	65.00
Express paint III	89.00
Forms in flight II	77.00
Future design 3-d	23.00
Graphic studio *	35.00
Human design 3-d	23.00
Icon Paint	59.00
Impact	55.00
Interchange	32.00
Invision	109.00
Lights Camera Action	49.00
Microbot design 3-d	23.00
Modeler 3d	64.00
Movie setter	65.00
Page flipper fx	95.00
Perfect vision	175.00
Photon expansion disk	19.00
Photon paint	65.00
Photon paint 2.0	97.00
Photon paint help	23.00
Photon video cell anim.	97.00
Pixmate	43.00
Print master plus	33.00
Prism	43.00
Pro video plus set I	83.00
Pro video plus set II	83.00
Pro. result using dpaint	35.00
Provideo plus	193.00
Sculpt 3-d	64.00
Silver turbo	129.00
The Director	46.00
Turbo silver	129.00
Tv-show	64.00
Tv-text	64.00
Video effects 3d	121.00
Video scape 3-d	120.00
Video titler	95.00
Zoetrope	91.00

DISKETTES

Fuji 3.5' ds/dd color disks	$19.00
Sony 3.5' ds/dd	19.00
TDK 3.5' ds/dd disks	19.00
Centech 3.5' ds/dd disks	19.00
Centech 5.25' ds/hd disks	29.00

ENTERTAINMENT SOFTWARE

We have them all! Over 300 games in stock. Call for your favorite game, and check our low prices.

SHOWROOM: 22 Front Street, Worcester, MA 01614

Serial Communications

High-performance direct-access routines

by Mike Schwartz
Copyright © 1989 Midnight Oil Software, Inc.

Mike Schwartz is a programmer who has written games for everything from coin-op arcade machines to the Commodore 64, Atari, and IBM PC. He started working with the Amiga in 1985, and now uses nothing else; he currently owns five Amigas. Mike is the author of ***Dark Castle*** *for the Atari and Amiga (he preferes the Amiga version), and is currently working on a sequel.*

Don't try this at home, kids! *In this article, and in future articles by Mike Schwartz, you will notice that non-standard programming practices are being used, such as accessing the Amiga's hardware directly. This should normally be avoided on the Amiga, since it conflicts with the multitasking system: if more than one program tries to use the serial port hardware directly, for example, there will be a conflict. Unless you are writing something like a game that completely takes over the system, you should never access hardware directly like this in a commercial program, since that would cause problems for users with other software, hurting not only the sales of your product, but the reputation of the Amiga as a multitasking machine. Of course, hackers always like to get to the bare metal and get maximum performance, and many programs never leave their own hard disk. So in that spirit, we present this low-level look at serial communications.*

The Paula chip in the Amiga features a very powerful serial communications port, which you can use to communicate to other computers or modems, serial printers, or other serial devices. Before going into greater detail about how to work the Amiga's serial port, a general discussion of serial communications is in order.

In standard RS232 connections, there is typically a DB25 connector, like the one on the back of your Amiga, which is connected to another DB25 connector on whatever your Amiga is talking to via a serial cable. Serial communication is accomplished by synchronizing the two communicating devices at a common *baud rate* and then sending data one bit at a time from one device to another. One pin on the DB25 connector of the Amiga is a DATA OUT line and another is a DATA IN line. Data can be coming and going at the same time, since there are separate lines in each direction.

When talking between a computer and a modem, you will use a standard serial cable. Pin 1 on the connector on one end of your cable is connected to Pin 1 on the connector on the other end of your cable. When you talk between a computer and another computer, you need a special kind of cable called a *null modem* cable. This type of cable connects pin 2 on one connector to pin 3 on the other and pin 3 to pin 2. Basically, pins 2 and 3 are crossed so that the IN from one computer is connected to the OUT of the other and vice versa. You can get null modem cables ready-made in the store, or you can get a null modem adapter, which is a small device that plugs into one end of your cable and crosses pins 2 and 3.

The baud rate at which two devices communicate indicates the speed at which the communcations take place. A rate of 2400 baud means that 2400 bits per second can be sent in one direction (or 4800 bits in both). For each byte that is transmitted, there is a 'start' bit and a 'stop' bit, for a total of 10 bits. Thus, at 2400 baud, 240 bytes per second can be sent in one direction, or 480 bytes in both.

The serial port on the Amiga is capable of 65536 theoretical baud rates, including the most important ones: 110, 300, 1200, 2400, 9600, 19200, and 57600. The maximum baud rate possible when communicating between two Amigas is over 200K baud, and we will see how this is done shortly. But first, let's take a look at the serial port hardware registers and interrupts.

The serial port can be operated in two possible modes: polled, and interrupt-driven. In interrupt-driven serial communications, you will need to write interrupt handlers to handle incoming bytes and outgoing bytes. When doing polled serial communications, the CPU must periodically check the RBF and TBE bits in the INTREQ register of the Paula chip. There are benefits and drawbacks to both methods.

There are two registers in the Paula chip that are used to hold the data bytes to be sent or received. When receiving a byte, Paula reads the start bit, the 8 data bits (which are copied into the Receive Buffer Register), and the stop bit, then sets the RBF bit in INTREQ to indicate that a byte is received. When sending a byte, Paula sends a start bit, the 8 data bits from the

Transmit Buffer Register, and the stop bit, then it sets the TBE bit in INTREQ. If the RBF and/or TBE bits are set in INTENA, an interrupt will also be generated. The amount of time required to send or receive a byte (plus start/stop bits) is called a 'byte time' and the time required for one bit is called a 'bit time'.

There are only a few hardware registers needed to perform serial I/O. SERPERW is used to set the baud rate. SERDATR is used to read the Receive Buffer register and status bits, and SERDATW is used to write to the Transmit Buffer register. The hardware designers of the Amiga made writing serial software a bit easier by 'mirroring' the TBE and RBF bits in SERDATR. This means that two of the bits in SERDATR always reflect the states of the TBE and RBF bits in INTREQ. Instead of reading two hardware registers (INTREQ and SERDAT) you only have to read one. The upper byte of SERDATW is used to set the size of the serial data (the Amiga's serial port can send 8 or 9 bits at a time) and also must have the stop bit set in it in order for transmission to work correctly. We will not consider the 9-bit mode here, since it is not of real value in any of our discussions.

The following example shows how to do polled serial I/O at 2400 baud:

```
BAUD        equ     2400                ; your desired baud rate here
BAUD_PERIOD equ     (3579545/BAUD)-1    ; value to store in SERPERW

SERDATR     equ     $dff018
SERDATW     equ     $dff030
SERPERW     equ     $dff032


init_serial move.w      #$4000,INTENAW  ; DISABLE!
            move.w      #BAUD_PERIOD,SERPERW
            move.w      #$3fff,$bfd0fe  ; init handshake lines
            move.w      #$0801,INTENAW  ; turn off RBF and TBE INTENA
            move.w      #$0801,INTREQW  ; turn off RBF and TBE INTREQ
            move.w      #$c000,INTENAW  ; ENABLE!
            rts

; outputs byte in D0 to serial port.  Uses d1.

rs232_out move.w        SERDATR,d1
          btst          #13,d1              ; check TBE
          beq.s         rs232_out           ; Transmit buffer busy?

; format the word to be written to SERDATW in d1

          move.w  #$100,d1            ; set stop bit, 8 bit data mode
          move.b  d0,d1               ; lower 8 bits = data
          move.w  d1,SERDATW
          move.w  #$001,INTREQW       ; turn off TBE bit
          rts

; reads byte from serial port into d0.b (upper bits are random)

rs232_in  move.w  SERDATR,d0
          btst    #14,d0
          beq.s   rs232_in

; thanks to the MIRRORed bit feature, the valid data
; is in d0.b!

          move.w  #$0800,INTREQW
          rts
```

Some comments before we go on. The above example is compatible with all versions of the Amiga operating system that I have been able to get my hands on. This means that you can use this code to drive the serial port instead of using the 'serial.device' that is part of the ROM Kernel. The benefits are quite obvious. When you use serial.device, you have to set up message ports and IORequest blocks. Serial.device also causes a lot of needless overhead because it is another task that is running in your system (once you open it). When you communicate with the serial.device, Exec is going to do a task switch, which is far more code than the above example requires. As a matter of fact, the serial.device had to be modified in version 1.2 of the operating system to provide a method of querying it to find out how many characters it had buffered so you could read them all with one DoIO() call. Reading one character at a time was too slow (due to task switching, etc.) to be used by application programs. *(See cautionary note at the beginning of the article - Ed.)*

The drawback to doing polled serial I/O is that if you do not read the Receive Buffer register often enough, a receive buffer overflow condition can occur. When a second byte comes in before the first one is read out of the Receive Buffer register, you are going to drop a byte of incoming data – an undesirable occurence! When an overflow occurs, bit 15 of SERDATR will be set. If you cannot guarantee that your code can read the Receive Buffer fast enough to prevent overruns, you should use interrupt-driven serial I/O, at least for input.

In order to implement interrupt-driven serial I/O, a little bit of programming strategy is in order. Instead of polling for input bytes, we are going to let Paula interrupt the CPU when the RBF flag is set. The interrupt handler for the serial port will read the Receive Buffer register and store the input data byte in a 'circular queue' for later access by your program. You can still get an overflow condition if your circular queue fills up. If your program is running in a multitasking environment, it may be suspended while other programs get to run. During this time, your serial interrupt routine will still be executed and will put bytes in the queue. When your program gets swapped in, you should be able to read enough bytes from the queue before you get swapped out again.

Interrupt-driven output also requires a queue for bytes that are to be output. Your TBE interrupt handler takes a byte from the output queue and outputs it to the serial port. Your *rs232_out* routine should put the output byte at the end of the queue if the queue is not empty or the Transmit buffer register is busy, or should output the byte to the hardware directly. You see, when you get a TBE interrupt and the queue is empty, the TBE handler won't have any data to send and will return right away. Since the Transmit buffer has emptied and generated its interrupt, there will be no more interrupts generated until your *rs232_out* routine outputs a byte directly to the hardware.

More on circular queues: A circular queue is an array of bytes. As bytes are added to the queue, they are added to the 'tail' of the queue. As bytes are removed from the queue, they are re-

moved from the 'head' of the queue. When the head and tail of the queue are the same, the queue is empty. Whenever the head or tail hits the end of the queue, it is pointed back at the start of the queue, thus making it circular. If the tail pointer wraps around and 'laps' the head pointer, the circular queue overflows.

Here is a complete interrupt-driven serial port driver example:

```
BAUD        equ     2400                ; your desired baud rate here
BAUD_PERIOD equ     (3579545/BAUD)-1    ; value to store in SERPERW

SERDATR     equ     $dff018
SERDATW     equ     $dff030
SERPERW     equ     $dff032

Level1AutoVec  equ     $64              ; Interrupt level 1
Level5AutoVec  equ     $74              ; Interrupt level 5

tbe_queue   ds.b    4096                ; enough for 4096 byte-times
tbe_head    dc.l    tbe_queue
tbe_tail    dc.l    tbe_queue

tbe_handler move.l      a0,-(sp)        ; two move.l to -(sp) is faster
            move.l      d0,-(sp)        ; than movem.l of two registers!

            move.l      tbe_head,a0
            cmp.l       tbe_tail,a0
            beq.s       99$             ; queue empty?

            move.w      #$100,d0        ; set stop bit!
            move.b      (a0)+,d0        ; get byte to send
            move.w      d0,SERDATW      ; output the byte
            cmpa.l      #tbe_queue+4096,a0
            ble.s       10$             ; no wrap?
            lea         tbe_queue,a0    ; yes wrap
10$         move.l      a0,tbe_head     ; update head pointer

99$         move.l      (sp)+,d0
            move.l      (sp)+,a0
            move.w      #1,INTREQW      ; ack TBE interrupt
            rte

rbf_queue   ds.b    4096                ; enough for 4096 byte-times
rbf_head    dc.l    rbf_queue
rbf_tail    dc.l    rbf_queue

rbf_handler move.l      a0,-(sp)        ; two move.l to -(sp) is faster
            move.l      d0,-(sp)        ; than movem.l of two registers!

            move.w      SERDATR,d0
            move.l      rbf_tail,a0
            move.b      d0,(a0)+

            cmpa.l      #rbf_queue+4096,a0
            ble.s       10$             ; no wrap?
            lea         rbf_queue,a0    ; yes wrap
10$         move.l      a0,rbf_tail     ; update head pointer

99$         move.l      (sp)+,d0
            move.l      (sp)+,a0
            move.w      #$800,INTREQW   ; ack rbf interrupt
            rte

init_serial move.w      #$4000,INTENAW  ; DISABLE!
            lea         tbe_handler,a0
            move.l      a0,Level1AutoVector
            lea         rbf_handler,a0
            move.l      a0,Level5AutoVector
            move.w      #BAUD_PERIOD,SERPERW
            move.w      #$3fff,$bfd0fe  ; init handshake lines
            move.w      #$8801,INTENAW  ; turn on RBF and TBE INTENA
            move.w      #$0801,INTREQW  ; turn off RBF and TBE INTREQ
            move.w      #$c000,INTENAW  ; ENABLE!
            rts

; outputs byte in D0 to serial port.  Uses d1.
rs232_out   move.w      #$4000,INTENAW      ; DISABLE
            move.l      tbe_tail,a0
            cmpa.l      tbe_head,a0
            beq.s       20$                 ; buffer empty...

5$          move.b      d0,(a0)+            ; put byte at tail of queue
            cmpa.l      tbe_queue+4096,a0   ; check for wrap
            ble.s       10$                 ; if no wrap
            lea         tbe_queue,a0        ; wrap
10$         move.l      a0,tbe_tail         ; update tail pointer
            move.w      #$c000,INTENAW      ; ENABLE
            rts

20$         move.w      SERDATR,d1
            btst      #13,d1                ; check TBE
            beq.s     5$                    ; Transmit buffer busy?

            ; format the word to be written to SERDATW in d1
            move.w      #$100,d1            ; set stop bit, 8 bit data mode
            move.b      d0,d1               ; lower 8 bits = data
            move.w      d1,SERDATW
            move.w      #$001,INTREQW       ; turn off TBE bit
            move.w      #$c000,INTENAW      ; ENABLE
            rts

; reads byte from serial port into d0.b (upper bits are random)
rs232_in    move.l      rbf_head,a0
            cmpa.l      rbf_tail,a0
            beq.s       rs232_in            ; wait for byte in queue
            move.b      (a0)+,d0            ; fetch data byte
            cmpa.l      #rbf_queue+4096,a0
            blt.s       10$
            lea         rbf_queue,a0
10$         move.l      a0,rbf_head
            rts

; checks for input byte ready
; returns NE if byte ready or EQ if byte not ready
rs232_csts  move.l      rbf_head,a0
            cmpa.l      rbf_tail,a0
            rts
```

I have been able to get two Amigas to communicate at speeds of four times the usual 'maximum' of 57600 baud. To get this speed, simply set the BAUD equate to this value in your software for both machines. At such a high baud rate, the Amiga's 68000 will have a hard time keeping up if you use interrupt-driven I/O. At this speed, there are over 20K bytes being sent or received in a second. When you consider how much overhead 20,000 interrupts take, you realize that your program will spend almost all of its time doing interrupts. At very high baud rates, I suggest that not only do you do polled I/O, but that you do it with *all* interrupts disabled. An example of this will be given in a future article on remote debugging.

Calling All User Groups!

Did you know that you can get volume discounts on subscriptions to *Transactor for the Amiga*?

For more information, call:

(416) 764-5273

or write:

Subscriptions Dept.
Transactor for the Amiga
85 West Wilmot St.
Richmond Hill, Ontario
Canada. L4B 1K7

NEW- F-BASIC™ V2.0 -NEW

F-BASIC™ is an Enhanced, Compiled Basic Language System.

- ★ Easy For Beginners
- ★ SO Fast—Beats C And Other Basics!
- ★ Recursive Subprograms
- ★ Over 350 Powerful Commands
- ★ Easy Access To Amiga Libraries
- ★ Comprehensive Users Manual
- ★ Easy Access to Amiga Windows, Menus, Sound, Events & Animation
- ★ Complete For Serious Programmers
- ★ Direct IFF File Support
- ★ Record Structures And Pointers
- ★ Ultra Fast Floating Point
- ★ Sample Program Disk—Over 100 Examples
- ★ Snobol-Like Pattern Matching

The **F-BASIC™** System Also Has A Source Level Debugger

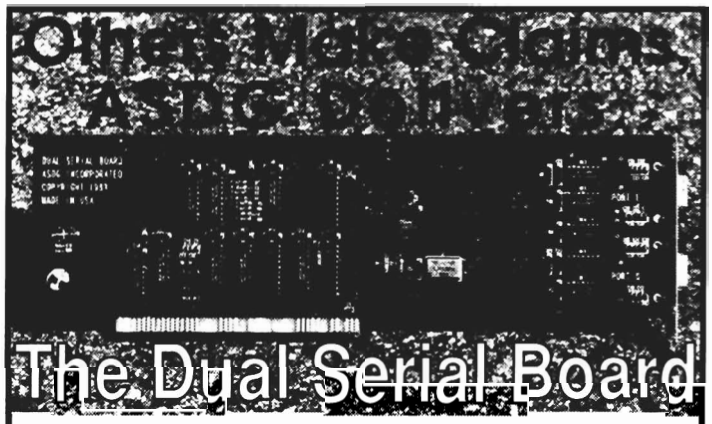
- ★ Debug **F-BASIC** Programs At The Source Level—So Easy.
- ★ Intuition Interface—Windows For Source, Registers, Memory, Variables.
- ★ Comprehensive Users Manual—Full Documentation.

The **F-BASIC™** Language System 2.0—Only $89.95
F-BASIC™ System and Complete SLDB—Only $149.95

Available Only From The Manufacturer
SEND CHECK OR MONEY ORDER TO:

DELPHI NOETIC SYSTEMS, INC.
Post Office Box 7743
Rapid City, South Dakota 57709-7743
Credit Card or C.O.D. Call (605) 348-0791

F-BASIC and FastCom are registered trademarks of DNS, Inc.
AMIGA is a registered trademark of Commodore/AMIGA Inc.



Fast, Compatible, Complete!

● Two high speed RS-232C serial ports for the Amiga 2000. ● IBM PC AT style (full 7 wire plus RI and DCD) connector for each port. ● Send and Receive at 110 to 115200 baud. ● Complete Amiga system software (both Exec *and* DOS). ● Using our exclusive Serial DisPatcher, *any* program which uses "serial.device" can use any port *without* patches. ● Public domain and shareware software on the distribution diskette. ● One year warranty.

ASK YOUR DEALER!

ASDG Incorporated
925 Stewart Street
Madison, WI. 53713
(608) 273 - 6585

Amiga is a trademark of Commodore-Amiga, Inc. IBM PC AT is a trademark of IBM Corp.

Random Number Generation

Creating random numbers - and testing them

by Eric Giguère
Copyright © 1989 Eric Giguère

A recent project of mine required the use of random numbers to generate fractal patterns on my Amiga's display. This got me wondering how accurate my compiler's random number generator was.

Random numbers play an important part in many computer applications ranging from games to simulations of real-life events. In the latter case it is extremely important to use a 'good' generator to ensure the conclusions based on a simulation's results are not incorrectly based on one particular pattern of random numbers.

Then again, a 'good' generator should be able to reconstruct any previous series of random numbers for debugging purposes.

As it turns out, the random number generator I was using – the one that comes with the Aztec C compiler – is a 'good' generator. It is also, unfortunately, unportable – the Lattice C compiler has its own generator. It would be nice if a portable random number generator was available for use with any Amiga C compiler.

What are random numbers?

Computer users use the term 'random numbers' fairly loosely to mean a sequence of numbers that satisfy certain special criteria.

First, that the numbers are *random*. There is no pattern to the order in which the numbers appear.

Second, that the numbers are *independent*. Independent is used here in the statistical sense to mean that the probability of any specific number showing up in the sequence is unaffected by the numbers that have previously been generated or will be generated in the future.

Third, that the numbers are *uniformly distributed* over a given range. In other words, each random number falls within a specific range and each number in that range will occur in the sequence as often as any other number. A cautionary note here: 'as often' refers to a sequence of *infinite* length; in any finite sequence it is likely that certain numbers will outnumber others.

As you may have realized by now, no digital computer can fulfill the first two criteria to their full extent. A computer is, after all, a *deterministic* machine – given the current state and a set of input to process or instructions to execute, the next state is always predictable. Even a hung machine is in a known state. (Small comfort!) So how can your Amiga generate a set of fully random and independent numbers?

Easy: we fake it and use pseudo-random numbers instead.

Pseudo-random numbers

A sequence of *pseudo-random numbers* is one that mimics a random sequence by relaxing the first two criteria for random numbers.

First, we substitute "there is no pattern to the order in which the numbers appear" with "there is no *discernible* pattern to the order in which the numbers appear". We don't care if the sequence eventually repeats itself, only that the length of the sequence between repeats is sufficiently large enough that we can't tell when the repetition will occur. We want the sequence to have a large *period*, as large as we can make it.

Second, we define 'independent' to mean that there is no *obvious* relationship between any two numbers in a sequence. What constitutes an 'unobvious' relationship is unfortunately unclear, but it should be one that the user can decipher only with great difficulty.

From now on we will use the term 'random numbers' to refer to pseudo-random numbers instead.

Good versus bad

A truly random sequence of numbers, one that satisfies the original set of criteria, would obviously be a 'good' set of numbers. What makes a set of pseudo-random numbers 'good' or 'bad'?

As it turns out, there's been a considerable amount of research into this very question. One of my references for this article, Donald Knuth's *The Art of Computer Programming,* dedicates the first 170 pages of Volume 2 to this very subject alone, mostly on proofs and mathematical discussions of the sort very few people enjoy. (A much more readable treatment of the subject – without the mathematics – can be found in Chapter 35 of Robert Sedgewick's *Algorithms.*)

Out of this research came several mathematical tests designed to weed out the 'good' random number generators from the 'bad' ones. The simplest to implement is known as the *chi-square* test.

The chi-square test is a simple statistical method for testing the uniformity – how well-spread the numbers are – of a sequence. For any given sequence of numbers in the range 0 to *r-1*, where *r* is some positive integer, the chi-square value for that sequence should be no greater than or no less than two times the square root of *r* away from *r*. This should be true eight or nine times out of ten to ensure that the numbers are uniformly spread across the range 0 to *r*.

Two implementations of the chi-square test – one written in C, the second in ARexx – are shown in Listings 1 and 2. The C file can be used to test any random number generator of the form:

```
unsigned long random( rmax )
unsigned long rmax;
```

where random() will generate a random integer from 0 to *rmax-1*.

Simply pass *rmax* and a pointer to the function to be tested to the routine ChiTest(). It will return FALSE if the generator fails the chi-square test more than two times out of ten. (Some random number generators generate floating-point values from 0 to 1 instead of integers. To test these, simply write a function that first calls the random number generator, multiplies the returned value by *rmax*, and then truncates it into an unsigned long.) The ARexx version performs the chi-square test on ARexx's built-in generator (though it is much much slower than the C version because ARexx is an interpreted language).

Results from the chi-square test show that Aztec's random number generator rand() passes the test, but that the ARexx random number generator RANDU() fails it, unfortunately.

The chi-square test is not the only test that should be performed, but the others are beyond the scope of this article. Use the chi-square test whenever you suspect the random number generator you are using might be faulty.

As another precaution, in any important simulation it's probably a good idea to use two or more different random number generators. A bad random number generator is worse than no random number generator.

Random number generation

There are two main methods to generate sequences of pseudo-random numbers, the *linear congruential* and *additive congruential* methods. The former is the method I used to develop a portable random number generator for the Amiga. The latter involves shifting and was discussed in some detail in an article by Gregory Knox in Volume 7, Issue 6 of the original *Transactor* and will not be discussed here. If you're interested, you can also look it up in Sedgewick's book.

The premise of the linear congruential method is very simple: take the previous random number, multiply it by a constant, add a constant and reduce the whole expression modulo some other constant. In C notation we can express this as

```
x = ( a * x + b ) % m;
```

where *a*, *b* and *m* are integer constants and *x* is the previous random number. This sequence must first be *seeded* before it can be used – that is, *x* has to be set to some non-zero value. We'll talk more about seeding below.

The linear congruential method is commonly used to generate random numbers because of its simplicity and speed. The question is, what values do we choose for *a*, *b* and *m*? The answer is far from obvious, but Knuth gives several guidelines in his book and several examples of 'good' generators that pass the tests for random number generators. The portable generator described in this article is based on such an example.

(For the curious: *m* should be chosen to be a convenient number close to the word size, as it determines the maximum period of the sequence, though the value of *a* can still reduce it significantly. *a* and *b* should be chosen to be relatively prime – setting *b* to 1 is the simplest method here. *a* requires a lot of fiddling to make sure the generated sequence is well-behaved – see Knuth for details.)

Why use the linear congruential method? Because it is simple and efficient. If *m* is chosen to be the *word size* of the machine (the largest integer a chip register can hold), and overflows are ignored, then it can be implemented using the expression:

```
x = a * x + b;
```

since ignoring any overflow makes the modulus irrelevant.

Seeds

The starting value for a random number sequence is called the *seed* value and it must be set by the user before a random number generator is called. The seed value should be different for each normal run of a program to yield a different random sequence. (Though when debugging it is sometimes useful to use the same seed value in order to repeat program behaviour and isolate certain bugs.)

To be useful, then, the seed value of a random number generator must itself be random – a Catch-22 situation. Popular

schemes involve asking the user for a seed number or using the current value of an I/O register as a seed value. Another method, using the Amiga system timer, is described below.

Bounding random numbers

The linear congruential method generates a fairly uniform sequence of integers in the range 0 to *m-1*. It's often the case that random integers are required to be lower than some upper bound *r*, where *r* is less than *m*. The obvious thing to do here would be to reduce the random number generated by the sequence modulo r, as in:

```
y = NextRandom() % r;
```

There is a problem with this in that the low-order digits of a sequence of numbers generated using the linear congruential method do not vary as uniformly as one would think, and so the numbers generated using the modulus operation are not as good as they should be.

A better method involves returning a random floating-point number *x/m* from 0 to 1 instead of returning a random number *x* from 0 to *m-1*. This number can then be multiplied by *r-1* to result in a 'better' random number in the range 0 to *r-1*. Unfortunately the conversions between integer and floating-point arithmetic slow things down appreciably, and also require the linking of a math library with the program. The portable generator uses a simpler method involving only integer division.

A portable random number generator

Listing 3, *random.c*, shows the C source code for a portable random number generator. (By 'portable' I mean usable with either the Aztec or Lattice C compilers.)

The basic functions are:

SeedRandom(seed) – seeds the random number generator

LRandom(max) – returns a random integer from 0 to *max-1*

The arguments to both and the return value for LRandom() are all of type unsigned long (32-bit integer). Remember to cast the parameters as unsigned long when you call the functions if you are using the 16-bit (default) option with Aztec C.

The random number generator is built around the linear congruential method using values chosen from Knuth's examples.

The function DRandom() returns a floating-point value between 0 and 1; it takes no parameters. (If you have no need for this function, simply comment it out – otherwise, you will be required to link with the math library after compilation.)

The portable generator also tries to provide a simple solution to the seeding problem with the SeedWithTime() function, which takes a single unsigned long as a parameter. SeedWithTime() uses the current microseconds value of the system timer added to the user-supplied value to seed the random number generator. This is a workable solution because in a multitasking, interactive system like the Amiga, the chances are very small that the seed value will ever be the same when a program is started up.

The source code should be easily translatable into other languages such as Modula-2, providing care is taken to handle the overflow situations which C ignores. Sedgewick's book provides useful information on how to do this.

Final notes

Random numbers can be very useful in many situations providing you choose your generator carefully. Always try a variety of generators whenever accurate results are important.

* * *

References

Donald E. Knuth, *The Art of Computer Programming, Volume 2*, Second Edition, Addison-Wesley.

Robert Sedgewick, *Algorithms,* Second Edition, Addison-Wesley.

Listing 1: *chisquare.c*

```
/*
 * File:  chisquare.c
 *
 * Written for the Transactor for the Amiga by Eric Giguere.
 * This code is in the public domain.
 *
 * Both functions in this file use a random number generator of the
 * form
 *
 *     unsigned long (*random_function)( unsigned long max );
 *
 * which returns an integer random number from 0 to max-1.  The generator
 * is expected to have been seeded before any tests are performed.
 */

#define NO_CHISQUARE_PROTOTYPES

#include "chisquare.h"

extern double sqrt();

/*
 * External functions -- this program uses malloc() and free()
 * instead of AllocMem() and FreeMem() for portability across
 * machines.
 */

extern void *malloc();
extern void  free();

/*
 * ChiSquare -- Perform the chi-square test on a random number generator.
 *              Returns the chi-square value for the given number of
 *              random numbers from 0 to r-1.  This value should be in
 *              the range ( r - 2 * sqrt(r) ) to ( r + 2 * sqrt(r) )
 *              8 times out of ten for the random number generator to
 *              be useful.  Returns -1000.0 if an error occurs.
```

```
 */

double ChiSquare( n, r, fp )
  unsigned long  n,      /* number of times to perform the test */
                 r,      /* upper bound on random number generation */
                (*fp)(); /* random number generator to be tested */
  {
    double         t, fi, b;
    unsigned long  i;
    unsigned long *f;

    /* Get memory for our array */

    f = (unsigned long *) malloc((unsigned)((r+1L) * sizeof(unsigned long)));

    if( f == NULL )
        return( -1000.0 );

    /* Clear the array */

    for( i = 0L; i < r; ++i )
        f[ i ] = 0L;

    /* Generate n random numbers and calculate their frequency */

    for( i = 0L; i < n; ++i )
        f[ (*fp)(r) ]++;

    /* Calculate the chi-square test value */

    b = ((double) n) / ((double) r);

    for( t = 0.0, i = 0L; i < r; ++i )
      {
        fi = ((double) f[i]) - b;
        t  = t + fi * fi;
      }

    free( f );

    return( t / b );
  }

/*
 * ChiTest -- Test a random number generator using the chi-square test
 *            above.  Returns TRUE if the generator passes an acceptable
 *            number of tests, FALSE otherwise.
 */

int ChiTest( r, fp )
  unsigned long  r,      /* upper bound on random numbers */
                 (*fp)(); /* random number generator to be tested */
  {
    double        sqrt_of_r, min, max, chi;
    unsigned long n;
    int           i, out_of_bounds;

    /* Calculate bounds for chi-square test */

    sqrt_of_r = 2.0 * sqrt( (double) r );

    min = (double) r - sqrt_of_r;
    max = (double) r + sqrt_of_r;

    n             = 15L * r;
    out_of_bounds = 0;

    /* Perform test 20 times -- values should be out
       of bounds at most four times */

    for( i = 0; i < 20; ++i )
      {
        chi = ChiSquare( n, r, fp );

        if( chi < min || chi > max )
            ++out_of_bounds;
      }

    if( out_of_bounds > 4 )
        return( FALSE );

    return( TRUE );
  }
```

Listing 2: *chisquare.rexx*

```
/*
 * File: chisquare.rexx
 *
 * Written for the Transactor for the Amiga by Eric Giguere
 * This code is in the public domain.
 *
 * This file tests the AREXX random number generator RANDU()
 * for its 'goodness' with respect to the chi-square test.
 *
 * Use: simply run this file passing it a number between 0 and 1
 *      to use as the seed value for the random number generator.
 */

/* Make sure the math library is loaded */

check = addlib( "rexxmathlib.library", 0, -30, 0 )

Signal on break_c
numeric digits 12

/* Define the parameters for our search */

n = 500
r = 50
lower_bound = r - 2 * sqrt( r )
upper_bound = r + 2 * sqrt( r )

say 'Chi-Square Test for Random Number Generators'

/* Parse the argument list */

parse arg seed remainder

if( seed = '' | remainder ~= '' )then
    call Usage

if( datatype( seed, Whole ) ~= 1 )then
    call Usage

/* Seed the random number generator */

x = randu( seed )

/* Run the test 20 times */

out_of_bounds = 0

say 'Chi-square values should be in the range'
say lower_bound 'to' upper_bound'.'

do i = 1 to 20
    chi = ChiSquare( n, r )

    say '    Chi-square value' i '=' chi
```

```
    if( chi < lower_bound | chi > upper_bound )then
        out_of_bounds = out_of_bounds + 1

    if( out_of_bounds > 4 )then
        break
  end

say 'Out of bound values =' out_of_bounds

if( out_of_bounds > 4 )then
    say 'Random number generator failed the test.'
else
    say 'Random number generator passed the test.'

Exit 0

break_c:

   Exit 1

/*
 * ChiSquare -- Calculate a chi-square value
 */

ChiSquare: procedure
  parse arg n, r

    b = n / r
    t = 0.0

    /* Clear our array */

    do i = 0 to r-1
        f.i = 0
      end

    /* Now run the test */

    do i = 1 to n
        x   = Rand( r )
        f.x = f.x + 1
      end

    /* Calculate the chi-square value */

    do i = 0 to r-1
        fi = f.i - b
        t  = t + fi * fi
      end

  Return( t / b )

/*
 * Rand -- Uses RANDU to generate a random integer
 *         from 0 to max-1
 */

Rand: procedure
  parse arg max
  Return( ( randu() * (max-1) ) % 1 )

/*
 * Usage -- Print usage information for this program
 */

Usage: procedure
    say
    say '  Usage: rx chisquare [seed]'
  Exit 1
```

Listing 3: *random.c*

```
/*
 * random.c
 *
 * Written for the Transactor for the Amiga by Eric Giguere.
 */

#include "exec/types.h"
#include "exec/devices.h"
#include "devices/timer.h"

static unsigned long RandomNumber;

/*
 * SeedRandom -- Seed the random number generator.
 */

void SeedRandom( seed )
  unsigned long seed;
  {
    RandomNumber = seed;
  }

/*
 * NextRandom -- Generate an random integer using the linear
 *               congruency method.  The parameters were chosen
 *               to follow Knuth's guidelines in Vol. 2 of
 *               The Art of Computer Programming.  The formula
 *               used is of the form
 *                       x = ( a * x + 1 ) mod m
 *               where m = 2^32 is the word size and so we
 *               let normal binary addition handle the modulus.
 */

#define A 1664525L

unsigned long NextRandom()
  {
    RandomNumber = A * RandomNumber + 1L;

    return( RandomNumber );
  }

/*
 * LRandom -- Returns a random integer between 0 and max-1.
 */

#define WORD_SQRT 65536L

unsigned long LRandom( max )
  unsigned long max;
  {
    return( ( ( ( NextRandom() / WORD_SQRT ) * max ) / WORD_SQRT ) );
  }

/*
 * DRandom -- Returns a random number between 0 and 1.
 */

#define WORD_SIZE 4294967296.0

double DRandom()
  {
    return( ((double) NextRandom()) / WORD_SIZE );
  }

/*
 * SeedWithTime -- Amiga-specific routine to seed the random number
```

```
 *                generator using the current microseconds value
 *                of the system timer.  Also accepts a parameter
 *                which can be added to this value to form a new
 *                seed value.
 */

void SeedWithTime( seed )
  unsigned long seed;
  {
    struct timerequest tr;

    /* Open the timer device -- if error then just seed
       using the given value */

    if( OpenDevice( TIMERNAME, UNIT_MICROHZ, &tr, 0L ) )
      {
        SeedRandom( seed );
        return;
      }

    /* Get the system time */

    tr.tr_node.io_Message.mn_Node.ln_Type = NT_MESSAGE;
    tr.tr_node.io_Message.mn_Node.ln_Pri  = 0;
    tr.tr_node.io_Message.mn_Node.ln_Name = NULL;
    tr.tr_node.io_Message.mn_ReplyPort    = NULL;
    tr.tr_node.io_Command                 = TR_GETSYSTIME;
    DoIO( &tr );

    SeedRandom( tr.tr_time.tv_micro + seed );

    CloseDevice( &tr );
  }
```

Listing 4: *chisquare.h*

```
#ifndef CHISQUARE_H_INCLUDED

    #define CHISQUARE_H_INCLUDED

    #include <stdio.h>

    #ifndef FALSE
        #define FALSE 0
        #define TRUE  1
    #endif

    #ifndef NO_CHISQUARE_PROTOTYPES

        #ifdef __STDC__
            extern double ChiSquare( unsigned long n, unsigned long r,
                                     unsigned long (*fp)( unsigned long ) );
            extern int    ChiTest( unsigned long r,
                                   unsigned long (*fp)( unsigned long ) );
        #else
            extern double ChiSquare();
            extern int    ChiTest();
        #endif /* __STDC__ */
    #endif  /* NO_CHISQUARE_PROTOTYPES */

#endif /* CHISQUARE_H_INCLUDED */
```

Listing 5: *random.h*

```
#ifndef RANDOM_H_INCLUDED

    #define RANDOM_H_INCLUDED
    #ifdef __STDC__
        extern unsigned long LRandom( unsigned long max );
        extern double        DRandom( void );
        extern void          SeedRandom( unsigned long seed );
        extern void          SeedWithTime( unsigned long seed );
    #else
        extern unsigned long LRandom();
        extern double        DRandom();
        extern void          SeedRandom();
        extern void          SeedWithTime();
    #endif

#endif
```

Listing 6: *testrand.c*

```
/*
 * This program tests the Aztec and Portable
 * random number generators.
 */

#include <stdio.h>
#include "chisquare.h"
#include "random.h"

/* Aztec routines */

extern int rand();
extern int atoi();

unsigned long aztecrand( max )
  unsigned long max;
  {
    return( (unsigned long) rand() % max );
  }

/*
 * The first argument should be a seed value
 */

main( argc, argv )
  int argc;
  char *argv[];
  {
    int i, seed;

    if( argc != 2 )
      {
        puts( "Usage: testrand [seed]" );
        exit(1);
      }

    seed = atoi( argv[1] );

    SeedRandom( (unsigned long) seed );
    srand( seed );

    puts( "Testing the Portable generator..." );

    if( ChiTest( 201L, LRandom ) == FALSE )
        puts( "Portable failed..." );

    puts( "Testing the Aztec generator..." );

    if( ChiTest( 201L, aztecrand ) == FALSE )
        puts( "Aztec failed..." );

    puts( "Done" );
  }
```

Colour Cycling

Simple animation for your C programs

by Hobie Orris

Hobie Orris is a software engineer living in Toronto who likes to program on his Amiga a lot more than he did on his old Apple II+.

In December 1985, I bought my Amiga 1000 and received a complimentary copy of Deluxe Paint. I wasted no time in assembling the system, booting up DPaint and loading every picture on the disk. One of the things I discovered that day was that by displaying the waterfall picture and hitting the TAB key, the river would appear to move and flow over the falls. I was impressed. I knew what colour cycling was because of a computer graphics course I had taken, but I had never before seen it in action. Since that time, colour cycling has become a familiar sight on the Amiga in paint programs, games and even disk copying programs.

Before we delve into the process of programming colour cycling, we should take a look at the underlying mechanisms that make it possible. As everyone who has used an Amiga paint program or written an Amiga graphics program knows, the colour information of your Amiga screen is not in the screen memory itself, but in a table of colour registers which interpret the screen pixel values into colour levels for the red, green and blue guns in the Amiga monitor. For example, the Workbench screen is two bitplanes deep, making possible four colours corresponding to the first four colour registers. This means that changing the colour of an area on the screen involves altering a single 16-bit colour register instead of the tens or hundreds of bytes of screen data that make up that area.

The cycling effect is achieved by defining a range of colour registers and rotating their values. The default low-res palette in Deluxe Paint II has a cycle defined for registers 20 to 31, which are 12 shades of gray proceeding from dark to light. When the tab key is pressed, the colours advance along the sequence, with the darker grays moving into the next higher slots and the lighter ones moving behind to replace them. This cycle will repeat continuously until you hit the tab key again to cancel colour cycling. All that is taking place here is that the values in the colour registers which define the screen colours are being rotated in place at a predefined speed. The simplicity and low processing overhead involved in changing colours makes colour cycling an attractive method of providing animation effects in a program at minimal cost to the programmer in time and effort.

Now this is just great, you are probably thinking, but how can I get that smoothly synchronized colour rippling effect when my program is busy getting joystick events and moving spaceships and heroes around the screen at breakneck speed? Well, things get even better because your program does not even have to shift a bit to accomplish perfectly synchronized cycling; all it needs to do is set up a hardware interrupt server, and let that server do all the work.

Hardware interrupts on the Amiga cause running programs to be interrupted in order for the processor to respond to some external stimulus, such as disk or keyboard input, copper or blitter operations, or the system clock. When an interrupt occurs, the processor enters supervisor mode and jumps to an interrupt service routine which takes care of the event that triggered the interrupt, and then returns to user mode and normal processing.

There are actually two different ways of treating an interrupt in the Amiga: an interrupt handler and an interrupt server chain. An interrupt handler is a routine which takes exclusive control of the interrupt processing, after which normal processing is resumed and the interrupt is cancelled. An example of a handler would be a serial input interrupt handler which transfers data from the serial port to an internal buffer. An interrupt server chain is a way of sharing an interrupt between various tasks. It consists of a list of routines which are executed in priority sequence before normal processing resumes. By inserting your server routine in the chain, you can accomplish what you want to do without having to take exclusive control of the interrupt.

For colour cycling, the vertical blanking interrupt is most useful. Every 1/60th of a second, the video beam completes drawing the video display, then turns itself off and repositions at the top left corner in order to begin all over again. This delay (while the beam is off) is called the vertical blanking interval. Animation and video programs usually wait for this interval before changing the screen contents so that there is no flickering of the image caused by contention between the graphics

and display hardware. There is a hardware interrupt associated with the vertical blanking interval which is associated with a server chain, so by hooking a colour cycling routine into this chain, you can achieve perfectly synchronized colour cycling. This cycling routine need only shuffle around a copy of the screen's colour registers and load them back into the display registers according to the correct cycling rate.

The accompanying code samples show how to read colour cycle data from an IFF file, how to hook into the interrupt server chain, and how to actually perform the cycling itself. The server code comes from the IFF source code diskette (Fish Disk 64). It was originally written by Dan Silva, author of Deluxe Paint, so some Deluxe Paint conventions remain unchanged. The direction of the cycle, either from lower to higher registers or vice versa, is set in Deluxe Paint by a small arrow pointing either up or down. This setting corresponds to the RNG_REVERSE flag being set (up) or not set (down) in the cycle control structure. The RNG_ACTIVE flag indicates that the cycle is turned on. Setting and clearing this flag is a convenient way of turning on or off individual cycles without affecting the other ones.

The global variable *cycling* in the example program is useful for turning them all on or off at once. The cycling rates correspond to those generated by Deluxe Paint when the timing slider is moved about. The lowest possible cycling rate is RNG_NORATE (36), which is what you get when the slider is all the way to the left. This value indicates the cycle is turned off. The maximum value, with the slider all the way to the right, is OnePerTick (16384) which would correspond to a cycle change every 60th of a second. Each successive click on the slider increments the rate by a nonlinear value of four at the low end to a maximum of 512 on the high end. The net result in Deluxe Paint is a cycling interval between 6.8 seconds and 1/60th of a second.

A skeleton IFF reader is included to demonstrate how cycling can be implemented in a program. A convenient way to write an IFF reader is to process IFF chunks in a switch statement, indexed by chunk type. In the example reader, the details of processing most chunks are left out for the sake of brevity. The cycle chunk looks like this:

```
struct cycle_chunk
{
  char chunktype[4];
    /* this contains 'CRNG' to identify the chunk */
  long chunksize;      /* this will be 8 */
  WORD pad1;           /* reserved for future use */
  WORD rate;           /* colour cycling rate */
  WORD flags;          /* control settings */
  UBYTE low, high;
    /* low and high colour registers of the range */
}
```

The routine getcycle() reads this chunk and assigns its values to the cycle_control structure, which is identical to the chunk with the exception that the pad1 field is used as a counter to determine on which interrupt the cycle changes. When start_cycle() is called, it first makes a copy of the screen's palette. It then initializes an interrupt structure with pointers to the interrupt server routine, MyVBlank(), and the program data segment. The data segment pointer is initialized only if the program is compiled using a small data model, that is to say it uses 16-bit pointers addressed relative to a segment register (A4 in Manx). This is the default configuration for Manx, and one which saves some code since it doesn't generate 32-bit addresses. The last thing start_cycle() does is to install the interrupt server routine by means of a call to AddIntServer(), specifying interrupt 5, the vertical blank interrupt. One last thing to note is that MyVBlank() calls LoadRGB4() using the screen's ViewPort as a parameter, so the screen must already be open before start_cycle() is called, and it must remain open until stop_cycle() is called, or you will have a mystical experience courtesy of the Guru.

The glory of colour cycling need not be limited to a picture viewer. I have seen games in which it is used to make water ripple, control panel readouts flash and alien eyeballs pulse. It can also be used for more mundane applications, such as text highlighting in spreadsheets (lending added drama to being 'in the red'), terminal programs or disk utilities. It is also one of the cheapest effects, in terms of programming time and effort, that anyone can do to add a little visual appeal to his or her program.

```
/*
 * Here's a skeleton IFF viewer program with colour cycling added.
 * A large portion of the code is not included for brevity's sake,
 * so please use your imagination. Or get the Transactor disk.
 *
 * Scrounged from various sources and expanded or modified
 * by Hobie Orris.
 *
 */

/********** Some IFF housekeeping stuff **********/

/*  Useful macro from EA */
#define MAKE_ID(a, b, c, d)\
 ( ((long)(a)<<24) + ((long)(b)<<16) + ((long)(c)<<8) + (long)(d) )

/*  IFF types we may encounter  */
#define FORM MAKE_ID('F', 'O', 'R', 'M')
#define ILBM MAKE_ID('I', 'L', 'B', 'M')
#define BMHD MAKE_ID('B', 'M', 'H', 'D')
#define CMAP MAKE_ID('C', 'M', 'A', 'P')
#define BODY MAKE_ID('B', 'O', 'D', 'Y')
#define GRAB MAKE_ID('G', 'R', 'A', 'B')
#define SPRT MAKE_ID('S', 'P', 'R', 'T')
#define CAMG MAKE_ID('C', 'A', 'M', 'G')
#define CRNG MAKE_ID('C', 'R', 'N', 'G')
#define CCRT MAKE_ID('C', 'C', 'R', 'T')
#define DPPV MAKE_ID('D', 'P', 'P', 'V')

/* a way of getting the chunk type and size */

#define GET_CHUNK_HEADER(t,s) { fread(&t, sizeof(t), 1, iffile);\
        fread(&s, sizeof(s), 1, iffile);\
       }
/* some random things */
#define NO  0
#define YES 1
#define LOCAL static
#define IS_AZTEC /* Manx compiler at work */
```

```
/********** Colour cycling definitions **********/

/* Colour cycle control structure.  This is similar, but not identical
 * to, the CRNG chunk itself.
 */
struct cycle_control
{
    SHORT count;  /* used to keep track of cycle duration */
    SHORT rate;   /* the colour cycling speed */
    SHORT flags;  /* non-zero means this cycle is active */
    UBYTE low, high; /* the range of colours are defined by these */
};

/* cycle_control flags values */

#define RNG_ACTIVE  1 /* this indicates the cycle is turned on */
#define RNG_REVERSE 2 /* this indicates the cycle is going backwards */
#define RNG_NORATE 36 /* this is the lowest cycling value on the dpaint
        palette requester, equivalent to "no cycling" */

#define NUM_CYCLES 4 /* this is the number dpaint gives you arbitrarily */

/********** VARIABLES ***********/

/* The screen's colour table, as read from the IFF file */
UWORD Colours[32];

/* Here's where to put the colour cycling control information
 * once it's read from the IFF file.
 */
struct cycle_control cycles[NUM_CYCLES];

/* This can be used to turn on and off colour cycling after the
 * service routine is installed, without removing the interrupt server.
 */
short cycling = TRUE;

main(argc, argv)
int argc;
char **argv;
{
 if (argc != 2)
  exit(1);

 /* STEP 1: Open libraries and initialize.
  */
  open_stuff(); /* not included */

 /*
  * STEP 2: Read the IFF image, including getting the colour cycling
  *   information from the IFF file.
  */
 iff_reader(*++argv);

 /*
  * STEP 3: Install the vertical blank interrupt server
  */
 start_cycle();

 /* STEP 4: Wait for the user to hit a key or button.
  */
 get_imsg();

 /* STEP 5: Remove the interrupt server and clean up.
  */
 stop_cycle();
 close_stuff();
 exit(0);
}

/*
 * Here's a skeleton IFF reader (based on Leo Schwab's) just to show you
 * how to fit the colour cycling code into your own personal IFF reader.
 */

iff_reader(fname)
char *fname;
{
 ULONG chunktype, chunksize, formsize;
 short cycle_index = 0;

 if (!(fd = fopen(fname, "r")))
 {
  printf("File open failed.\n");
  return(-1);
 }

 /* check that this is an IFF file */
 GET_CHUNK_HEADER(chunktype, chunksize);
 if (chunktype != FORM)
  return(-1);

 formsize = chunksize;

 /* check that it is really an ILBM (InterLeaved BitMap) picture */
 fread(&chunktype, sizeof(ULONG), 1, iffile);
 if (chunktype != ILBM)
  return(-2);

 formsize -= sizeof(chunktype);

 /* Keep reading in chunks until the end of the FORM is reached */
 while (formsize > 0)
 {
  GET_CHUNK_HEADER(chunktype, chunksize);
  formsize -= sizeof(chunktype) + sizeof(chunksize);

  switch(chunktype)
  {
   /* bit map header (height, width, depth, screen position) */
   case BMHD:
    /* get the BitMapHeader */
    break;

   /* colour map */
   case CMAP:
    /* get the colour map and put it in the array Colours */
    break;

   /* bit plane data */
   case BODY:
    /* allocate, read in and uncompress the bitplanes */
    break;

   /* the colour cycling control chunks */
   case CRNG:
    if (cycle_index < NUM_CYCLES)
    {
     getcycle(iffile, &cycles[cycle_index]);
     cycle_index++;
    }
    else /* skip it */
     fseek(iffile, chunksize, 1);
    break;

   /* display modes */
   case CAMG:
    /* read in display mode flags, eg. HAM, HIRES, etc. */
    break;

   /* stuff I don't want to know about */
   case GRAB:
   case DPPV:
   case CCRT:
   case SPRT:
    fseek(iffile, chunksize, 1);
    break;

   default:  /* unknown chunk, skip it */
    fseek(iffile, chunksize, 1);
    break;
  }
  formsize -= chunksize;

  /* handle odd size chunks */
  if (chunksize & 1)
  {
   formsize--;
   fseek(iffile, 1L, 1);
  }
 }
}
```

```
/* Read in the CRNG chunk at which the file is positioned
 */
getcycle(file, cyc)
FILE *file;
struct cycle_control *cyc;
{
 fseek(file, 2L, 1); /* dump padding */
 cyc->count = 0; /* start off at beginning */
 fread(&cyc->rate, sizeof(WORD), 1, file);
 fread(&cyc->flags, sizeof(WORD), 1, file);

 /* actually 2 vals, low and high */
 fread(&cyc->low, sizeof(WORD), 1, file);
}


/*
 *  Dan Silva's DPaint color cycling interrupt code
 *    with some minor modifications.
 *
 *  Use this as an example for interrupt driven color cycling.
 *  If compiled with Lattice, use -v flag on LC2.
 *  If compiled with Manx, define IS_AZTEC.
 */


/* Cycling Frame Rates */
/* The maximum visible cycling rate available in dpaint is 16384 which
 * corresponds to once per screen refresh (60 per second) and the minimum
 * is 40, corresponding to once every 6.8 seconds.
 */
#define OnePerTick   16384
#define OnePerSec    OnePerTick/60 /* in case you wanted to know */

/* temporary storage for the colour register values as they are cycled */
UWORD cycols[32];

/* This is the vertical blank interrupt server.  Once it has been installed
 * by start_cycle() then every 60th of a second this routine is called.
 */
MyVBlank()
{
 int i,j;
 LOCAL struct cycle_control *cyc;
 LOCAL WORD temp;
 LOCAL BOOL anyChange; /* flag indicates whether the registers changed */

#ifdef IS_AZTEC
#asm
 movem.l  a2-a7/d2-d7,-(sp)
 move.l   a1,a4
#endasm
#endif

 if (cycling)
 {
  anyChange = NO;
  for (i=0; i<NUM_CYCLES; i++)
  {
   cyc = &cycles[i];

   /* ignore inactive cycles */
   if (cyc->low == cyc->high || !(cyc->flags & RNG_ACTIVE) ||
    cyc->rate == RNG_NORATE)
     continue;

   /* increment the cycle timer */
   cyc->count += cyc->rate;

   if (cyc->count >= OnePerTick)
   {
    anyChange = YES;
    cyc->count -= OnePerTick;

    /* if cycling in reverse */
    if (cyc->flags & RNG_REVERSE)
    {
     temp = cycols[cyc->low];
     for (j = cyc->low; j < cyc->high; j++)
      cycols[j] = cycols[j + 1];
     cycols[cyc->low] = temp;
    }
    else /* cycling forward */
    {
     temp = cycols[cyc->high];
     for (j = cyc->high; j > cyc->low; j--)
      cycols[j] = cycols[j - 1];
     cycols[cyc->low] = temp;
    }
   }
  }
  if (anyChange)
     LoadRGB4(VP,cycols,32);
 }

#ifdef IS_AZTEC
#asm
 movem.l  (sp)+,a2-a7/d2-d7
#endasm
#endif

 return(0);  /* interrupt server routines should do this */
}


/*
 *  Code to install/remove cycling interrupt handler
 */

LOCAL char myname[] = "MyVB";  /* Name of interrupt handler */
LOCAL struct Interrupt intServ;

typedef void (*VoidFunc)();

/* Install the vertical interrupt server.  This only needs to be done once.
 */

start_cycle()
{
 APTR GetA4();

 /* Copy the screen's colour table into the cycling storage area */
 movmem(Colours, cycols, 64);

#ifdef IS_AZTEC
 intServ.is_Data = GetA4();  /* returns contents of register a4 */
#else
 intServ.is_Data = NULL;
#endif
 intServ.is_Code = (VoidFunc)&MyVBlank;
 intServ.is_Node.ln_Succ = NULL;
 intServ.is_Node.ln_Pred = NULL;
 intServ.is_Node.ln_Type = NT_INTERRUPT;
 intServ.is_Node.ln_Pri  = 0;
 intServ.is_Node.ln_Name = myname;
 AddIntServer(5L,&intServ);
}

/* This removes the interrupt server routine from the server chain.
 * Do this before your program exits or you will have something to meditate
 * about.
 */
stop_cycle()
{
 RemIntServer(5L,&intServ);
}

#ifdef IS_AZTEC
/*
 * Return the contents of register A4 which points to the Manx data segment.
 * This allows the server routine to make use of variables belonging to the
 * host program.  If compiled with the large code or data options, this is
 * not necessary.
 */
#asm
 public _GetA4

_GetA4
 move.l a4,d0
 rts
#endasm
#endif
```

Toy: A Small Compiler For The Amiga

Part 3: Parsing

by Chris Gray

Chris Gray is a programmer from Edmonton, Alberta, and currently works for Myrias Reasearch Corporation, which is bulding a parallel processing supercomputer. He has programmed the Amiga since its beginning, and has released such products as the shareware Draco compiler, along with its accompanying libraries and tools, and Amiga Empire.

This third article in the series on the Toy compiler describes the parsing routines. These are divided up into three modules. The first deals with procedure headers, along with the entry and exit code used, and with variable declarations. The second deals with expressions, and the third deals with statements. As has been mentioned previously, the Toy compiler does not build an internal structure to represent a parsed program. Thus, the parsing routines call on the code generation routines directly. I have tried to keep the parser relatively independent of the target processor, but since I haven't tried to port Toy to a different CPU, I don't really know how successful I have been. Perhaps an ambitious MS-DOS programmer will see these articles and build an 8086 version!

The parsing technique used is 'recursive descent'. This means, quite literally, that the parser is recursive, and it descends into the structure of the program. The file handling code calls the 'top' levels of the parser, which handle things like global variables and procedures. They in turn descend into lower levels, like statements and expressions. Places in which the grammar loops (a right-hand-side refers back, either directly or indirectly, to its own left-hand-side) correspond to a loop or recursion in the parsing routines.

The reader may wish to quickly glance back at the previous article, to see the interfaces defined for the symbol table and tokenizer code.

Declarations and Procedures

Listing 1 shows the file *decl.d*. It contains only two functions, *declareVariables* and *defineProc*. The first is used to define a group of variables, either global to the program, or local to a procedure. The second is used to define a procedure, and to call the code generation routines at the right places to cause the procedure's entry and exit code to be built. This second is one of the larger procedures in the compiler. For a more complex language, both of these would be split up into more pieces. In particular, there would be a function to parse an arbitrary type and return some kind of structure defining it. Also, most languages treat two descriptions of the same type as equivalent, so there would be facilities for either merging multiple descriptions into one, or of comparing two descriptions. To illustrate this, consider the following Draco-style declarations:

```
[10] * char fred;
[10] * char joe;
```

Here, both 'fred' and 'joe' are the same type – array of 10 pointers to character – thus we can assign one to the other, or compare the two. In order to allow this, the compiler has to either use the same structure (typically an array node, which has an element-type which is a pointer node, which in turn has a subtype which is a built-in type node) for both, or to provide a routine which traverses these structures and decides they are equivalent. Avoiding this whole area is the main reason I kept Toy down to just 'int' and 'bool' as types.

declareVariables is given one parameter, which is the kind of variables to be declared. This is either *sk_localVariable* for variables local to a function, or *sk_globalVariable* for variables global to the entire program. This value is stored in the symbol table entries created for the variables, and is also used by the code generation routines to assign storage to the variables. The structure of *declareVariables* is that of two nested **while** loops. The inner one loops through a declaration of a number of variables of the type, and the outer one loops past several of the inner sets. For example, in:

```
int a, b, c, d;
bool fred, joe, sam, harry;
bool alice, mary, sue, evangeline;
```

there would be three traversals of the outer loop, each consisting of four traversals of the inner loop.

When we enter the function, the current token will be the keyword for the type, which is either 'int' or 'bool'. We thus arrange for the outer loop to continue while that situation is true.

This means that the body of that loop must consume all legal tokens up to and including the semicolon that separates the declaration sets. We remember the type in local variable 'typ' ('type' is a reserved word in Draco, so I can't use that as the variable name), then move past the token. At that point we expect to see a list of undeclared identifiers, separated by commas, followed by a semicolon. We arrange the inner loop so that it always starts looking at the new identifier. *getToken* is called instead of *getSimpleToken*, since we need the symbol pointer in order to define the symbol. If the identifier is already defined, we complain about it and go on. Otherwise, we give it the appropriate kind and type, then call *allocateVariable* in the code generation portion of the compiler to let it figure out where the variable will be. It will use the symbol's *se_value* field to store any needed information.

After skipping the newly defined identifier token, we run head on into one of the uglier parts of most compilers – that of error recovery. The Toy grammar expects a comma or a semicolon after the identifier. What happens if there isn't one, but there is, say, a '<' token instead? Well, we have to put out an error message, and then try to continue meaningfully. Good error recovery is a careful balance between skipping too much input (which could result here in some variables not getting declared, which would cause extra error messages when those variables were used) and skipping too little input (which could result in extra error messages here, and perhaps the compiler getting into an infinite loop if it ends up in the same place again). I won't try to justify the recovery that Toy does – it is a result of my experience with tinkering with compilers.

For correct input, the inner loop would have skipped a comma, and would now be looking at the next identifier, or would have found a semicolon and done nothing about it, thus leaving it for the outer loop. The outer loop expects the semicolon between declaration sets, and thus has a very similar error-recovery situation. Here the recovery is dependent on the kind of the variable – if we are inside a procedure, we should stop at anything that looks like an executable statement, but if we are not, we should stop at anything that looks like a new declaration or a procedure definition.

Declaration processing is one area that is more complicated in a C-type language. In Draco, even the most complicated type is handled by the general type parser, and variable declaration is very much as is shown here. In C, however, the type is a combination of the basic type given at the beginning, and the modifiers given around the identifier ('*' before it for pointer, parentheses and bounds lists after it for function types and arrays). Because surrounding parentheses are allowed, the identifier is in fact not seen until the bottom of a recursive routine! Parts of the type structure are built bottom-up, and other parts are built top-down. In my ANSI-C compiler, I found that good error recovery required some finely tuned code with arbitrary token lookahead.

defineProc is responsible for all of the work of parsing a procedure definition. This includes handling its parameter list, its result type, any local declarations, its body, and any work needed to help the code generation routines produce correct entry and exit code. When it is called (by the system interface code), the current token will be the 'proc' token. The first step then, is to skip that token. After the 'proc' should be an undeclared identifier – the name of the new procedure. Here we see another aspect of error recovery – that of preventing the compiler from crashing when given invalid input. Later portions of *defineProc* modify the procedure's symbol table entry. If no identifier was given, we don't have a symbol table entry. Thus, either those places must all check for this case, or we should build a fake one to use. The latter option is used here.

After the procedure name should be a left parenthesis. Next are the procedure's parameters. These are handled very similarly to the declaration of variables. The error recovery is slightly different yet again, and the symbol kind used here is *sk_parameter*. We also restrict the parameters to be of type 'int', so that the symbol table entry for the procedure need contain only the number of parameters. In general, it would contain something like a linked list of parameters and their associated types. After any parameter (there may be none) should be the closing right parenthesis.

It is easy to miss the very important calls to the code generator here. *procStartParameters* is called before declaring any parameters; *procEndParameters* is called after the parameters but before any local variables; and *procEndLocals* is called after the local variables. *declareVariable* is called for each parameter and local variable. This interface should be adequate for any target CPU, and could even allow the code generator to decide to keep some variables in registers.

In Toy, procedure *main* is special – it is called by the startup code to start the user's program. Thus, it has no parameters, and also returns no result. C's *main* is more complex – it is passed a description of the command-line arguments to the program, and returns a status result. Draco handles this with library calls, so presumably an expanded Toy would do the same.

After giving the procedure symbol its proper kind and parameter type, we look for a procedure result type. That is also put into its symbol table entry. We then expect to see a colon, marking the end of the procedure header. After getting the local variables, we can call upon the code generation routines to get ready to generate code for the procedure's body. It is on this call (to *codeInit*) that the procedure's entry code would be produced. This entry code must save any registers used by the procedure, allocate stack space for local variables, and make the local variables and parameters accessible to the body code.

The part of defining a procedure that seems most important to a programmer, that of handling the body of the procedure, is surprisingly simple – it just calls on the statement parser to handle a list of semicolon-separated statements. Error recovery here consists of continually calling *parseStatementList* until a 'corp' or the end of the input file is found. By convention, *parseStatementList* returns the type of the last statement or ex-

pression in the list. Error checking here is making sure that the type returned matches what is expected based on the result type of the procedure.

procResult is yet another special entry in the code generator which does whatever is necessary to make the last used expression the result of a current procedure. *codeTerm* then does the procedure exit code and any other cleanup needed. *symbolPurge* is then called to remove all local definitions from the symbol table. This will include the local variables and parameters of the procedure, but not the procedure itself. The procedure ends with a **corp** and an optional semicolon.

Expressions

Listing 2 shows *expression.d*, the Toy expression parser. In this file we will start with the last function and work backwards to the front. *parseExpression* is the external interface to the expression parser. It is used by the statement parser in several places. It does nothing other than call *parseComparison*, which is the top level of the recursive descent expression parser. This level of indirection allows for the addition of more levels of expressions, such as boolean operators, which would have lower precedence than the comparison operators.

parseComparison is typical of the expression handling routines. It first calls the next lower level of the expression parsing chain, in this case *parseAddSub*. It then examines the next input token. While that token is a comparison operator, it loops, parsing and generating code for a comparison. When an appropriate operator is found, the previously obtained subexpression, which is to be the left-hand operand for the operator, is checked for an appropriate type. The operator token is then skipped (skipping the operator after the type test associates an error message with the left-hand operand, rather than with the operator).

Next, the lower level routine, *parseAddSub*, is called to get the right-hand operand. That operand is also checked to make sure it is appropriate. For a more complex language, the check would have to compare the types of the left and right-hand operands for comparison compatibility.

When all is well, the code generator is called on to generate a comparison (*compareValues*). It is passed a saved copy of the operator token, so that it knows which type of comparison to use. Note that there are no other calls on the code generator – the operands are set up by the lower level parts of the expression parser. The result of the function, like all of these functions, is the type of the expression. If a comparison was done, the type is 'bool', otherwise the type is that of the subexpression that would have been the left-hand side if a comparison was present.

The next routine (moving backwards), *parseAddSub* is entirely analagous. Beyond is *parseMulDiv*, again quite similar. Note that these last two share the same code generation routine. Next comes *parseUnary*, which is a little different – it uses an **if** instead of a **while**, and calls itself recursively if it finds a unary operator. At this point it is useful to go through an example parse, to see how all of the cases fit together. We will parse the expression:

```
1 + 2 * -+3 - +++++4 < 5 / (6 + 7)
```

When *parseExpression* is called, the current token will be the '1'. The chain of calls will immediately go down through *parseComparison*, *parseAddSub*, *parseMulDiv*, and *parseUnary*, landing in *parseUnit*, which will handle the number and return. The current token will then be the '+', which will allow the return to go back up to *parsePlusMinus*, which will enter its while loop. It will check the type of what is now its left-hand operand, which is 'int', therefore okay. Skipping over the '+', it will dive back down the chain again. *parseUnit* will again handle the number (2), and return back up, this time to *parseMulDiv*. *parseMulDiv* will move past the '*' operator, and call *parseUnary* for its right-hand operand. This time, *parseUnary* sees one of the operators it is looking for (the unary '-'), so it goes into its larger alternative. The operator is saved and skipped, and *parseUnary* is called recursively. This time it sees the unary '+', which is handled the same way. The next call to it sees no unary operator, so *parseUnit* is called to handle the number 3. We now return to the lower of two recursive *parseUnary* calls and check the operand type. All is well, but nothing is done, since the operator was a unary '+', which is a do-nothing in Toy. So, we return to the upper of the recursive *parseUnary* calls. This time the operator was unary '-', and we call the code generation routine *unaryOp*, giving it the '-' token. This is the first call to the code generator, other than calls done by *parseUnit*, even though we have processed seven expression tokens. Other operations are remembered in the local states of the parsing routines. The recursion in *parseUnary* is done so that the code generation is done in the proper order. For unary operators, the execution is done right-to-left, whereas the operators are seen left-to-right. Using recursion allows us to hold an arbitrary number of them pending.

The return from the uppermost *parseUnary* call goes back to the second call of *parseUnary* inside *parseMulDiv*. After a type check, it will finally call *binaryOp* to do the multiply of 2 and -3. Its while loop will now exit, and we return to *parseAddSub*, which will be in the same situation – it will call *binaryOp* to do the add. The **while** loop is again satisfied by the '-', so we go down for another operand. Note what is happening here – the binary operators are being evaluated left-to-right, as they should. *parseComparison* does the same thing, even though, for example, `a < b ~= c` is not valid. This is done for error recovery purposes.

Continuing with our parse, *parseUnary* will recurse its way through the string of unary '+'s, doing nothing with them. After the usual handling of the '4', we return back to *parseAddSub* which can complete the handling of the subtraction. We finally return to *parseComparison*, which sees a comparison operator and dives back down for its right-hand operand. The difference here is the parenthesized subexpression. We get down to *parseUnit* as usual, but here some-

thing special happens. It sees the '(', skips it and then calls *parseExpression*, which is the top of the chain. This call is the reason for the "recursive" in the name "recursive descent". Parsing of the subexpression '6 + 7' is standard until we come to the ')'. It matches none of the operator tests, so the calls return all the way out of *parseExpression*, and we end up back in *parseUnit*. Now we check for and skip the ')', and return, thus treating the parenthesized sub-expression just like any other bottom-level unit.

Some observations about the recursive descent parsing method:

- no temporary storage is needed - everything is kept in local variables in the parsing routines
- there is no limit to the size or complexity of the expression parsed, other than the size of the execution stack for the compiler
- operator precedence or priority is determined by the position of the handler for that operator in the chain of descending routines. Here the precedence goes

  unary to '*' and '/' to '+' and '-' to comparison
- new operators are added by augmenting an existing routine if they are at a previously used precedence, or by inserting a new routine in the call chain
- although Toy doesn't have any, postfix operators are added much like prefix (unary) operators

We have seen most of *parseUnit* except for a couple of details. The default in its if-else chain is where the compiler ends up if it gets confused. This case should consume at least one token, unless we are sure that we won't get back here without somebody else consuming one. Handling of identifiers is a bit special. When we see one at the beginning of a statement or expression, we don't know if it is being used as a value, or we have an assignment statement or a procedure call. Thus, we must skip the identifier and make that decision based on the next token. Even then, a procedure call could be a void procedure, which is a statement, or a valued procedure, which is part of an expression. Thus, the compiler treats statements and expressions much the same, except that statements must be of type 'void'. *parseUnit* uses a *pushXXX* code generation call to tell the code generator about new bottom-level operands. The model is that of a stack machine – push an operand, push another operand, perform the operation. It is entirely up to the code generator as to just how it implements this, however. *checkUndef* is a utility routine that complains if the identifier is not defined, and sets its type to *rt_error*.

parseProcCall should by now be fairly straightforward. It does some checking to avoid excess error messages, and just calls *pushParameter* for each parameter to the routine being called. Note that the parameters would already be stacked, but this call is telling the code generator that they are to be used as parameters. Finally, *callProc* is called to generate the actual call (which would be a JSR or BSR for a 68000 CPU).

A useful feature of many compilers is that of constant expression evaluation, also called constant folding. Here, any operator for which both operands are constants is performed directly by the compiler, instead of generating code to do the operation. It is of course done iteratively/recursively so that, for example, the above expression would end up as just '-9 < 0' or 'true'. This evaluation could be added to Toy as part of the parser, or as part of the code generator. In Draco, it is done during parsing – each routine of the chain returns not only the type of the expression, but also a description of where it is, which can be 'a constant whose value is XX'. The parsing routines thus check directly for two constant operands, and do the operation if needed. Doing it that way unfortunately blurs the separation between the parser and the code generator. A better way might be to insert a layer between the parser and the code generator which is just responsible for constant folding.

Statements

Listing 3 shows *statement.d*, the statement parser for Toy. After going through the expression parsing, most of it should be relatively straightforward. Once again, it is better to start at the end of the file and work back to the front. Recall that a procedure body is done by calling *parseStatementList*, which returns the type of the last statement/expression in the semicolon separated list. A semicolon after anything means that that thing must have been a statement. Toy, like Draco, will not accept an expression as a statement.

parseStatementList is a simple loop which calls *parseStatement* repeatedly, until it encounters a token which indicates the end of a statement list. The condition here is a negative one: "loop until you find one of these", whereas the loops in the expression parser were positive ones: "loop while you see one of these". This choice is made for error recovery reasons. *parseStatement* simply calls one of the handlers of specific statement types. Recall that assignment statements are handled by calling *parseAssignmentStatement* from inside the bottom level of the expression parser.

An assignment statement in Toy consists of a variable name, ':=', and an expression. *parseAssignmentStatement* calls *checkAssign* to verify that the given symbol is a variable that can be assigned to. *popVariable* tells the code generator to store the latest expression value into the indicated variable. The error checking done here simply ensures that the expression has the same type as the variable. No error message is given if either is *rt_error*, which indicates that there has been an error previously, and so we don't want an extra one now. In a more complex language, assignment compatibility would be considerably more complex.

A **write** statement in Toy consists of 'write', '(', a list of expressions or string constants, and ')'. This is in fact the only place in the compiler where the value of a string constant is

actually used. *isStatement* is a utility routine which returns 'true' if the current token can start a statement. *parseWriteStatement* loops through strings or expressions, calling special code generation routines to arrange for them to be printed. Here we see the first use of the type *RelocCode_t*, to tell *callSpecial* which of the special run-time routines is to be called.

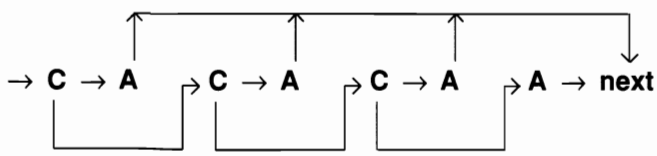
parseReadlnStatement is very similar, except that reads can only be done into assignable variables. The *checkUndef*, needed here, is not needed in *parseAssignmentStatement* since the check was already done in the expression parser. A new input line is read (callSpecial(rc_readLine)), and then the required number of integers are taken from that line (callSpecial(rc_readInt)).

parseWhileStatement is the first routine we have seen which needs to generate branches in the output code. The code generator interface to do this consists of four routines. *getPos* returns a value which can be used as the target of a later backwards branch. *branchTo* generates an unconditional backwards branch to a previously determined target. *branchForward* generates a conditional or unconditional branch to some forward location. It returns an identifier that can be used to fix up the branch when the target is known. *patchBranch* patches such a branch to branch to the current code position. This simple interface is adequate for Toy, but may not allow for the generation of good code. Toy's positions and identifiers are actually just offsets in the code for the current procedure.

Given the above, *parseWhileStatement* is quite simple – skip past the **while**, get a statement list for the condition, skip the **do**, get a statement list for the body, and skip the **od**. An unconditional branch is generated to go from the end of the body to the beginning of the condition statement list, and a conditional branch, which branches on a 'false' condition, is made from the end of the condition to just past the looping branch. If one wanted to cripple the **while** statement in Toy to requiring just a simple expression as the condition (thus disallowing centre and end-exit loops), just replace the first call to *parseStatementList* with a call to *parseExpression*.

parseIfStatement is more complicated. It allows the use of the **elif** form in **if** statements. This means that it must keep track of a number of places where the ends of the alternatives have to all branch to after the entire if statement. In Draco, these are all chained together in the code buffer (which also helps the forward-branch-shortening code to work), but I wanted a cleaner interface in Toy, so a simple fixed-size array is used. This limits the Toy compiler to a maximum number (ELIF_MAX) of **elif**s in a single **if** statement.

The generated code for an **if** statement with **elif**s looks something like this:



where the C's are conditions, the A's are alternatives, and 'next' is the statement that comes after the **if** statement. Each condition either drops into its alternative (the condition is 'true'), or branches to the next condition. The last condition branches to the **else** part if there is one. Each alternative branches to the next statement. Array 'offsets' contains the positions of each of the forward branches from the ends of alteratives to the next statement. The other branches are maintained by simple state variables, since there is only one of them active at a time.

The loop in *parseIfStatement* loops over all of the conditional tests and alternatives. The first is an **if**, and the remainder (if any) are **elif**s. The loop is a bottom-exit loop since we want to exit when we have fully processed an alternative, and the next token is not **elif**. The loop first skips the if/elif. On all except the first time around, it then handles the termination of the previous alternative. This involves generating an unconditional forward branch (to the next statement), and patching the previous condition's branch to come 'here'. Next, it gets the expression to be used as the condition for this alternative. This call to *parseExpression* could be replaced with a call to *parseStatementList*, but that could lead to some very ugly Toy code. Next comes a check for the 'then' token, followed by a call to *parseStatementList* for the body of the alternative. Finally we have the loop condition, along with some error checking. After the loop, we allow for a single 'else' part, which is very similar to the previous alternatives, except that it doesn't have a condition. After any 'else', all that is left is patching all of the end-of-alternative branches to come 'here'.

The interested reader might want to trace through the parsing of the example given in the first article, and reproduced here:

```
int N, Sum;

proc main()void:

    Sum := 0;
    readln(N);
    while N ~= 0 do
        Sum := Sum + N;
        readln(N);
    od;
    write("The sum is: ", Sum, "\n");
corp;
```

What sequence of calls to the code generator is produced?

Miscellaneous

Not listed here, but included on disk, is the file *parseUtil.d*, which includes some utilities used during parsing. *checkUndef* complains about an undeclared identifier, and sets it up as *sk_undefined*, with type *rt_error*. *checkAssign* complains if an identifier isn't a local or global variable or a parameter. *isStatement* returns 'true' if the current token can appear after a semicolon, i.e. is one that can start a new statement or is a separating token in a compound statement like **if** or **while**.

findStatement skips tokens until it finds such a token. It is used in error recovery. *findClose*, another error recovery routine, skips tokens until it finds a right parenthesis, a comma, or something that can start a statement.

Adding New Language Constructs

Since it is the parser that defines the language compiled, it is in the parser that changes are needed to add new language constructs. Adding new types, such as arrays, structures, pointers, etc. would be done either in *decl.d* or in a new file that just parses types. As an example, parsing a Draco-style array type consists of something like:

```
skip the '['
dimensions := 0;
while
    parseExpression();
    verify expression is compile-time integral constant
    dimensionTable[dimensions] := expression value;
    dimensions := dimensions + 1;
    token = ','
do
    skip the ','
od;
verify and skip a ']'
call 'parseType' to get the element type
```

As was mentioned previously, adding compound types to Toy would require the addition of type structures, better type checking, etc. Also required for array types would be array indexing in the expression parser. This would also have to be available for the left-hand-side of assignment statements. Also needed are the two new tokens, '[' and ']'.

A new construct, such as a **case** statement, would be done by adding a new routine to *statement.d*, along with a test for **case** and a call to the routine in *parseStatement*. A first decision here would be what the code for a case statement looks like. Most compilers use two (or more) variants. If the case indices are closely spaced, a table of code offsets is generated, and the code produced uses the case indexing expression value to index into that table to yield an offset to branch to. This may require setting up a 'default' case. If the case indexes are sparse, some sort of lookup is needed. Lattice C uses a linear search done directly inline. Draco calls a run-time-system routine to do a binary search of a sorted table. Other compilers might code a binary decision tree directly. Whatever methods are used, it is likely that additional entry points into the code generator would be needed. And of course, additional token types would be needed as well.

Next Month

Next month's article will discuss the code generator. We have seen most of its entry points in this article, but nothing of just what it does. We will go into the details of storage allocation and the construction of MC68000 instructions. Some trivial optimizations will be shown, and more details of the system interface will be discussed.

Listing 1: *decl.d*

The folowing program listings are in the *Draco* language. The complete Draco system can be found on Fish Disks 76 and 77.

```
#drinc:util.g
#toy.g
#externs.g

/*
 * decl.d: declaration and definition of the various symbols.
 */

/*
 * declareVariables - declare a set of variables of the given kind.
 */

proc declareVariables(SymbolKind_t kind)void:
    *SymbolEntry_t se;
    register TokenKind_t tk;
    ResultType_t typ;

    while
 tk := getSimpleToken();
 tk = tk_int or tk = tk_bool
    do
 typ := if tk = tk_int then rt_int else rt_bool fi;
 nextToken();
 while
     tk := getToken(&se, nil, nil);
     tk = tk_id
 do
     if se*.se_kind ~= sk_empty then
  errorHereThree("symbol '", se*.se_name, "' already defined");
     else
  se*.se_kind := kind;
  se*.se_type := typ;
  allocateVariable(se);
     fi;
     nextToken();
     tk := getSimpleToken();
     if tk = tk_comma then
 nextToken();
     elif tk ~= tk_semicolon then
 errorHere("expecting comma between variable names");
 while
      skipToken();
      tk := getSimpleToken();
      tk ~= tk_id and tk ~= tk_semicolon and tk ~= tk_comma and
  tk ~= tk_eof and tk ~= tk_int and tk ~= tk_bool
 do
 od;
 if tk = tk_comma then
     nextToken();
 fi;
    fi;
od;
if tk = tk_semicolon then
    nextToken();
else
    errorHere("expecting semicolon after variable list");
    if kind = sk_globalVariable then
 while
      skipToken();
      tk := getSimpleToken();
      tk ~= tk_semicolon and tk ~= tk_proc and tk ~= tk_eof and
  tk ~= tk_int and tk ~= tk_bool
 do
 od;
    else
 while
      skipToken();
      tk := getSimpleToken();
      tk ~= tk_semicolon and tk ~= tk_corp and tk ~= tk_id and
  tk ~= tk_if and tk ~= tk_while and tk ~= tk_readln and
  tk ~= tk_write and tk ~= tk_int and tk ~= tk_bool
```

```
  do
  od;
     fi;
     if tk = tk_semicolon then
  nextToken();
     fi;
 fi;
    od;
corp;

/*
 * defineProc - the outer part of defining a proc.
 */

proc defineProc()void:
    SymbolEntry_t dummyEntry;
    *SymbolEntry_t procSe, se;
    register TokenKind_t tk;
    register uint parCount;
    ResultType_t functionResult, bodyResult, typ;

    nextToken();
    tk := getToken(&procSe, nil, nil);
    if tk = tk_id then
 if procSe*.se_kind ~= sk_empty then
     errorHereThree("symbol '", procSe*.se_name, "' already defined");
     procSe := &dummyEntry;
     dummyEntry.se_name := " ";
 else
     /* put in non-empty kind, to prevent conflict with parameters */
     procSe*.se_kind := sk_undefined;
 fi;
 nextToken();
    else
 errorHere("expecting identifier as proc name");
 procSe := &dummyEntry;
 dummyEntry.se_name := " ";
 if tk ~= tk_leftParen then
     skipToken();
 fi;
    fi;
    tk := getSimpleToken();
    if tk = tk_leftParen then
 nextToken();
    else
 errorHere("expecting '(' to start parameter list");
    fi;
    procStartParameters();
    parCount := 0;
    while
 tk := getSimpleToken();
 tk = tk_int or tk = tk_bool
    do
 typ := if tk = tk_int then rt_int else rt_bool fi;
 if typ = rt_bool then
     errorHere("proc parameters can only be 'int'");
 fi;
 nextToken();
 while
     tk := getToken(&se, nil, nil);
     tk = tk_id
 do
     if se*.se_kind ~= sk_empty then
  errorHereThree("symbol '", se*.se_name, "' already defined");
     else
  se*.se_kind := sk_parameter;
  se*.se_type := typ;
  allocateVariable(se);
     fi;
     parCount := parCount + 1;
     nextToken();
     tk := getSimpleToken();
     if tk = tk_comma then
 nextToken();
    elif tk ~= tk_semicolon and tk ~= tk_rightParen then
 errorHere("expecting comma between parameter names");
 while
     skipToken();
     tk := getSimpleToken();
     tk ~= tk_id and tk ~= tk_semicolon and tk ~= tk_comma and
  tk ~= tk_eof and tk ~= tk_int and tk ~= tk_bool and
  tk ~= tk_rightParen and tk ~= tk_colon
 do
 od;
 if tk = tk_comma then
     nextToken();
 fi;
    fi;
od;
if tk = tk_semicolon then
    nextToken();
elif tk ~= tk_rightParen then
    errorHere("expecting semicolon after parameter list");
    while
 skipToken();
 tk := getSimpleToken();
 tk ~= tk_semicolon and tk ~= tk_eof and tk ~= tk_int and
     tk ~= tk_bool and tk ~= tk_rightParen and tk ~= tk_colon
    do
    od;
    if tk = tk_semicolon then
 nextToken();
    fi;
fi;
   od;
   procEndParameters();
   if tk = tk_rightParen then
nextToken();
tk := getSimpleToken();
   else
errorHere("expecting ')' after parameter list");
   fi;
   if parCount ~= 0 and CharsEqual(procSe*.se_name, "main") then
errorHere("'main' cannot have any parameters");
   fi;
   procSe*.se_kind := sk_procedure;
   procSe*.se_parCount := parCount;
   if tk = tk_int or tk = tk_bool then
functionResult := if tk = tk_int then rt_int else rt_bool fi;
if CharsEqual(procSe*.se_name, "main") then
    errorHere("'main cannot have a result");
fi;
nextToken();
   elif tk = tk_void then
functionResult := rt_void;
nextToken();
   else
functionResult := rt_error;
errorHere("expecting result type for proc");
   fi;
   procSe*.se_type := functionResult;
   tk := getSimpleToken();
   if tk = tk_colon then
nextToken();
tk := getSimpleToken();
   else
errorHere("expecting ':' after proc header");
   fi;
   declareVariables(sk_localVariable);
   procEndLocals();
   codeInit(procSe);
   while
if tk = tk_corp or tk = tk_eof then
    bodyResult := rt_void;
else
    bodyResult := parseStatementList();
```

```
    tk := getSimpleToken();
 fi;
 tk ~= tk_corp and tk ~= tk_eof
    do
 errorHere("expecting 'corp' to end proc");
 findStatement();
    od;
    if functionResult = rt_int or functionResult = rt_bool then
 if bodyResult = rt_void then
     errorHere("expecting result expression at end of function");
 else
     if functionResult ~= bodyResult then
  errorHere("type mismatch for proc result");
     fi;
     procResult();
 fi;
    elif functionResult = rt_void and
 (bodyResult = rt_int or bodyResult = rt_bool)
    then
 errorHere("unwanted result expression at end of procedure");
    fi;
    codeTerm(procSe);
    symbolPurge();
    if tk = tk_corp then
 nextToken();
    else
 errorHere("missing 'corp' on last proc");
    fi;
    if getSimpleToken() = tk_semicolon then
 nextToken();
    fi;
corp;
```

Listing 2: *expression.d*

```
#toy.g
#externs.g

/*
 * expression.d - routines for parsing, etc. expressions.
 */

/*
 * parseProcCall - parse a call to a procedure.
 */

proc parseProcCall(register *SymbolEntry_t se)void:
    register uint parCount;
    register TokenKind_t tk;
    register ResultType_t typ;
    bool errorFlagged;

    if getSimpleToken() = tk_leftParen then
 nextToken();
    else
 errorHere("expecting '(' to start proc parameters");
    fi;
    parCount := 0;
    errorFlagged := se*.se_kind ~= sk_procedure;
    while
 tk := getSimpleToken();
 tk ~= tk_eof and tk ~= tk_rightParen and
     (tk = tk_id or tk = tk_number or not isStatement())
    do
 typ := parseExpression();
 if typ = rt_void then
     errorHere("expecting expression as proc parameter");
 else
     if typ = rt_bool then
  errorHere("parameters must be of type 'int'");
     fi;
     pushParameter();
 fi;
 parCount := parCount + 1;
 if not errorFlagged and parCount > se*.se_parCount then
     errorFlagged := true;
     errorHere("too many parameters supplied");
 fi;
 tk := getSimpleToken();
 if tk = tk_comma then
     nextToken();
 elif tk ~= tk_rightParen then
     errorHere("expecting ',' between proc parameters");
     findClose();
 fi;
    od;
    if parCount < se*.se_parCount and se*.se_kind = sk_procedure then
 errorHere("too few parameters supplied");
    fi;
    if getSimpleToken() = tk_rightParen then
 nextToken();
    else
 errorHere("expecting ')' to end proc parameters");
    fi;
    callProc(se);
corp;

/*
 * parseUnit - bottom level of recursive descent expression parsing.
 */

proc parseUnit()ResultType_t:
    register TokenKind_t tk;
    *SymbolEntry_t se;
    ulong number;
    register ResultType_t typ;

    tk := getToken(&se, nil, &number);
    if tk = tk_id then
 checkUndef(se);
 nextToken();
 tk := getSimpleToken();
 if tk = tk_assign then
     parseAssignmentStatement(se);
     typ := rt_void;
 elif tk = tk_leftParen then
     parseProcCall(se);
     typ :=
  if se*.se_kind = sk_procedure then
      se*.se_type
  else
      rt_error
  fi;
 elif se*.se_kind = sk_undefined then
     typ := rt_error;
     pushConstant(0);
 else
     /* sk_globalVariable, sk_parameter or sk_localVariable */
     typ := se*.se_type;
     pushVariable(se);
 fi;
    elif tk = tk_number then
 nextToken();
 typ := rt_int;
 pushConstant(number);
    elif tk = tk_true or tk = tk_false then
 nextToken();
 typ := rt_bool;
 pushConstant(if tk = tk_true then 1 else 0 fi);
    elif tk = tk_leftParen then
 nextToken();
 typ := parseExpression();
 if getSimpleToken() = tk_rightParen then
     nextToken();
 else
     errorHere("missing matching ')' in expression");
 fi;
```

```
    else
 errorHere("syntax error - statement or expression needed");
 if not isStatement() then
     skipToken();
 fi;
 typ := rt_error;
 pushConstant(0);
    fi;
    typ
corp;

/*
 * parseUnary - parse the unary '+' or '-' operators.
 */

proc parseUnary()ResultType_t:
    register TokenKind_t tk;
    register ResultType_t typ;
    bool hadMinus;

    tk := getSimpleToken();
    if tk = tk_plus or tk = tk_minus then
 hadMinus := tk = tk_minus;
 nextToken();
 typ := parseUnary();
 if typ = rt_void then
     errorHere("can't use statement as operand");
 else
     if typ ~= rt_int and typ ~= rt_error then
  errorHere("operand to unary '+' or '-' must be int");
     fi;
     if hadMinus then
  unaryOp(tk);
     fi;
 fi;
 rt_int
    else
 parseUnit()
    fi
corp;

/*
 * parseMulDiv - parse the binary '*' and '/' operators.
 */

proc parseMulDiv()ResultType_t:
    register TokenKind_t tk;
    register ResultType_t typ;

    typ := parseUnary();
    while
 tk := getSimpleToken();
 tk = tk_star or tk = tk_slash
    do
 if typ = rt_void then
     errorHere("can't use statement as operand");
     pushConstant(0);
 elif typ ~= rt_int and typ ~= rt_error then
     errorHere("operand to binary '*' or '/' must be int");
 fi;
 nextToken();
 typ := parseUnary();
 if typ = rt_void then
     errorHere("can't use statement as operand");
 else
     if typ ~= rt_int and typ ~= rt_error then
  errorHere("operand to binary '*' or '/' must be int");
     fi;
     binaryOp(tk);
 fi;
 typ := rt_int;
    od;
    typ
corp;

/*
 * parseAddSub - parse the binary '+' and '-' operators.
 */

proc parseAddSub()ResultType_t:
    register TokenKind_t tk;
    register ResultType_t typ;

    typ := parseMulDiv();
    while
 tk := getSimpleToken();
 tk = tk_plus or tk = tk_minus
    do
 if typ = rt_void then
     errorHere("can't use statement as operand");
     pushConstant(0);
 elif typ ~= rt_int and typ ~= rt_error then
     errorHere("operand to binary '+' or '-' must be int");
 fi;
 nextToken();
 typ := parseMulDiv();
 if typ = rt_void then
     errorHere("can't use statement as operand");
 else
     if typ ~= rt_int and typ ~= rt_error then
  errorHere("operand to binary '+' or '-' must be int");
     fi;
     binaryOp(tk);
 fi;
 typ := rt_int;
    od;
    typ
corp;

/*
 * parseComparison - parse the comparison operators.
 */

proc parseComparison()ResultType_t:
    register TokenKind_t tk;
    register ResultType_t typ;

    typ := parseAddSub();
    tk := getSimpleToken();
    if tk = tk_equal or tk = tk_notEqual or tk = tk_less or
 tk = tk_lessEqual or tk = tk_greater or tk = tk_greaterEqual
    then
 if typ = rt_void then
     errorHere("can't use statement as operand");
     pushConstant(0);
 elif typ ~= rt_int and typ ~= rt_error then
     errorHere("operand to comparison must be int");
 fi;
 nextToken();
 typ := parseAddSub();
 if typ = rt_void then
     errorHere("can't use statement as operand");
 else
     if typ ~= rt_int and typ ~= rt_error then
  errorHere("operand to comparison must be int");
     fi;
     compareValues(tk);
 fi;
 rt_bool
    else
 typ
    fi
corp;

/*
 * parseExpression - top level of recursive descent expression parser.
```

```
 * This could have been just 'parseComparison', but doing it this
 * way allows better nameing and easier expansion.
 */

proc parseExpression()ResultType_t:

    parseComparison()
corp;
```

Listing 3: *statement.d*

```
#toy.g
#externs.g

/*
 * statement.d: parsing and code generation of statements.
 */

uint ELIF_MAX = 10;

/*
 * parseIfStatement - parsing and code generation for the 'if' statement.
 */

proc parseIfStatement()void:
    [ELIF_MAX + 1] uint offsets;
    register uint offsetCount, lastBranch;
    register TokenKind_t tk;
    register ResultType_t typ;
    bool first;

    first := true;
    offsetCount := 0;
    while
 nextToken();
 if not first then
     if offsetCount = ELIF_MAX + 1 then
  abort("too many 'elif's in 'if'");
     fi;
     offsets[offsetCount] := branchForward(false);
     offsetCount := offsetCount + 1;
     patchBranch(lastBranch);
 fi;
 first := false;
 typ := parseExpression();
 if typ = rt_void then
     errorHere("condition for 'if' must be expression");
 else
     if typ ~= rt_bool and typ ~= rt_error then
  errorHere("condition for 'if' must be bool");
     fi;
     lastBranch := branchForward(true);
 fi;
 if getSimpleToken() = tk_then then
     nextToken();
 else
     errorHere("expecting 'then' in 'if'");
 fi;
 if parseStatementList() ~= rt_void then
     errorHere("'if' expressions not allowed");
 fi;
 tk := getSimpleToken();
 if tk ~= tk_elif and tk ~= tk_else and tk ~= tk_fi then
     errorHere("expecting 'elif', 'else' or 'fi' in 'if'");
 fi;
 tk = tk_elif
    do
    od;
    if tk = tk_else then
 nextToken();
 if offsetCount = ELIF_MAX + 1 then
     abort("too many 'elif's in 'if'");
 fi;
 offsets[offsetCount] := branchForward(false);
 offsetCount := offsetCount + 1;
 patchBranch(lastBranch);
 if parseStatementList() ~= rt_void then
     errorHere("'if' expressions not allowed");
 fi;
 tk := getSimpleToken();
    else
 patchBranch(lastBranch);
    fi;
    while offsetCount ~= 0 do
 offsetCount := offsetCount - 1;
 patchBranch(offsets[offsetCount]);
    od;
    if tk = tk_fi then
 nextToken();
    else
 errorHere("expecting 'fi' to end 'if'");
    fi;
corp;

/*
 * parseWhileStatement - parsing and code generation for the 'while' statement.
 */

proc parseWhileStatement()void:
    register uint loopPos, branchPos;
    register ResultType_t typ;

    nextToken();
    loopPos := getPos();
    typ := parseStatementList();
    if typ = rt_void then
 errorHere("condition for 'while' must be expression");
    else
 if typ ~= rt_bool and typ ~= rt_error then
     errorHere("condition for 'while' must be bool");
 fi;
 branchPos := branchForward(true);
    fi;
    if getSimpleToken() = tk_do then
 nextToken();
    else
 errorHere("expecting 'do' in 'while'");
    fi;
    if parseStatementList() ~= rt_void then
 errorHere("body of 'while' must not be an expression");
    fi;
    branchTo(loopPos);
    patchBranch(branchPos);
    if getSimpleToken() = tk_od then
 nextToken();
    else
 errorHere("expecting 'od' to end 'while'");
    fi;
corp;

/*
 * parseReadlnStatement - handle the 'readln' statement.
 */

proc parseReadlnStatement()void:
    *SymbolEntry_t se;
    register TokenKind_t tk;

    nextToken();
    if getSimpleToken() = tk_leftParen then
 nextToken();
    else
 errorHere("expecting '(' in 'readln' statement");
    fi;
    callSpecial(rc_readLine);
    while getToken(&se, nil, nil) = tk_id do
 checkUndef(se);
```

```
 checkAssign(se);
 if se*.se_type ~= rt_int and se*.se_type ~= rt_error then
     errorHere("only int variables can be used in 'readln'");
 fi;
 callSpecial(rc_readInt);
 popVariable(se);
 nextToken();
 tk := getSimpleToken();
 if tk = tk_comma then
     nextToken();
 elif tk ~= tk_rightParen then
     errorHere("expecting ',' between variables in 'readln'");
     findClose();
 fi;
    od;
    if getSimpleToken() = tk_rightParen then
 nextToken();
    else
 errorHere("expecting ')' to end 'readln' statement");
    fi;
corp;

/*
 * parseWriteStatement - handle the 'write' statement.
 */

proc parseWriteStatement()void:
    *char buffer;
    ulong number;
    register TokenKind_t tk;
    ResultType_t typ;

    nextToken();
    if getSimpleToken() = tk_leftParen then
 nextToken();
    else
 errorHere("expecting '(' in 'write' statement");
    fi;
    while
 tk := getToken(nil, &buffer, &number);
 tk ~= tk_eof and tk ~= tk_rightParen and
     (tk = tk_id or tk = tk_number or not isStatement())
    do
 if tk = tk_string then
     emitString(buffer, number);
     callSpecial(rc_printString);
     nextToken();
 else
     typ := parseExpression();
     if typ = rt_void then
  errorHere("expecting string or expression in 'write'");
     else
  if typ ~= rt_int and typ ~= rt_error then
      errorHere("only int values can be used with 'write'");
  fi;
  callSpecial(rc_printInt);
     fi;
 fi;
 tk := getSimpleToken();
 if tk = tk_comma then
     nextToken();
 elif tk ~= tk_rightParen then
     errorHere("expecting ',' between expressions in 'write'");
     findClose();
 fi;
    od;
    if getSimpleToken() = tk_rightParen then
 nextToken();
    else
 errorHere("expecting ')' to end 'write' statement");
    fi;
corp;

/*
 * parseAssignmentStatement - handle ':='; variable already given.
 */

proc parseAssignmentStatement(register *SymbolEntry_t se)void:
    register ResultType_t typ;

    checkAssign(se);
    nextToken();
    typ := parseExpression();
    if typ = rt_void then
 errorHere("assignment right-hand-side must be expression");
    else
 if typ ~= se*.se_type and typ ~= rt_error and se*.se_type ~= rt_error
 then
     errorHere("assignment incompatibility");
 fi;
 popVariable(se);
    fi;
corp;

/*
 * parseStatement - handle a single statement or expression.
 */

proc parseStatement()ResultType_t:
    register TokenKind_t tk;
    ResultType_t typ;

    typ := rt_void;
    tk := getSimpleToken();
    if tk = tk_if then
 parseIfStatement();
    elif tk = tk_while then
 parseWhileStatement();
    elif tk = tk_readln then
 parseReadlnStatement();
    elif tk = tk_write then
 parseWriteStatement();
    else
 typ := parseExpression();
    fi;
    typ
corp;

/*
 * parseStatementList - handle a ';'-separated list of statements, which
 * can end in an expression.
 */

proc parseStatementList()ResultType_t:
    register TokenKind_t tk;
    register ResultType_t typ;

    typ := rt_void;
    while
 tk := getSimpleToken();
 tk ~= tk_eof and tk ~= tk_int and tk ~= tk_proc and tk ~= tk_corp and
     tk ~= tk_elif and tk ~= tk_else and tk ~= tk_fi and
     tk ~= tk_od and tk ~= tk_do
    do
 if tk = tk_semicolon then
     if typ ~= rt_void and typ ~= rt_error then
  errorHere("can't use expression as a statement");
     fi;
     nextToken();
     typ := rt_void;
 else
     typ := parseStatement();
 fi;
    od;
    typ
corp;
```

Introducing Icon

A superb string processing language

by Scott Ballantyne
Copyright © 1989 Scott Ballantyne

Scott Ballantyne is a programmer whose current interests are operating systems and computer language design. He has played a leading role in the ARP development effort since its inception; he wrote much of the arp.library code. Scott may be reached on CIS (70066,603) or BIX (sdb).

Icon has nothing to do with icons, and is not an acronym for anything.

Now that I have answered your first question, let me tell you a little more about it. It is a serious, general purpose, high-performance programming language, the syntax of which resembles a combination of C and Pascal. Its string handling surpasses the previous champion string processing language, SNOBOL4. Its innovative design and unique control structures make it easy to provide natural and compact solutions for many difficult yet common problems.

Icon's programmer friendliness and powerful trace facility make it an ideal language for quick one-shot programs or for prototyping large software systems. It is used for everything from enhancing or replacing shell scripts on Unix systems to prototyping neural networks at research facilities. Its features also make it an excellent choice for a first computer language.

Icon language features

Icon contains a wide variety of useful data structures including strings, tables, lists, sets, and a user definable data type called a record.

There are no type declarations in Icon; when necessary (and possible), Icon will convert from one data type to another automatically, which greatly simplifies programming. A surprising example is shown below, where we add a string literal to a floating point number and store the result in a variable:

```
s := "1.45" + 2.10
```

Icon also manages memory for you, including allocation, freeing and garbage collection. This makes a tremendous difference when using such data types as strings, tables or lists, the size of which can vary greatly during the course of a program.

Icon is a procedural language; in fact, programming with a large subset of Icon is not very different from programming in C or Modula-2. Unlike many languages, however, everything in Icon is an expression – there are no statements. Expressions in Icon may fail to produce any results; or they may generate huge quantities of results.

The simplicity of Icon, combined with the concepts of failure and multiple results from expressions, provides a powerful environment for controlling the complexity of programs. At the simplest level, for example, you can use the success or failure of a control expression to copy all lines in a file:

```
while line := read () do
   write (line)
```

The function read() reads an entire line from a file; write() writes it out. The *while* keyword repeatedly executes and evaluates read(), and, as long as read() succeeds, it executes the write() function. When read() fails, the *while* loop terminates.

Superficially, this appears to be no different from the boolean conditionals of other languages, and in fact this loop would be written similarly in C or Pascal. In these languages however, the read() function would have to return a value to indicate FALSE (typically, zero); a test of this true/false value is implicit in the control structures of these languages. Icon expressions don't return anything when they fail, and this allows some interesting program simplifications. For example, the loop above can be shortened to:

```
while write (read ())
```

When read() reaches the end of the file, it will fail (it won't return anything). Since write() now has nothing to display, it will also fail, and this failure will cause the *while* loop to be exited. This idiomatic form of the *while* loop, using the failure of one expression to cause the entire enclosing expression to fail, is common in Icon.

The expressions in Icon that can produce more than one result are called 'generators'. (Actually, *all* expressions in Icon are

generators – Icon's designers are big fans of consistency.) The success or failure outcomes of generators make the evaluation of Icon expressions goal-directed. This appears in contexts such as the following:

```
find ("i", "This instance of incendiary optimism") > 15
```

Find() is one of Icon's string functions. It searches its second argument for the presence of its first argument, and if it finds it, it returns the numerical position at which the first argument was found. The expression above is asking if there is an occurrence of the string "i" beyond position 15 in the string. Clearly, in this example, there are many such occurrences; this expression will produce a result in Icon where it would fail in most conventional programming languages.

Icon evaluates the expression above as follows: find() produces its first result, three, and then the comparison is performed. Since three is less than 15, the comparison fails; this would end the matter in languages like Modula-2 or C. However, in Icon, find() is fired up again, right where it left off, and produces its next result, a six. The comparison once again fails, and find() is re-executed for what turns out to be the final time, generating an 18. Since 18 is greater than 15, the comparison succeeds at this point and the expression terminates. Find() is not called again, although there are many more instances of "i" that have positions greater than 15.

This is a much more natural expression of this problem than is possible with conventional languages, which require loops and many auxiliary variables that have nothing to do with the actual problem.

You can also cause generators to produce all their possible values by using the *every* keyword. For example,

```
every write (find ("oo", "Too Soon Too Say Good Bye"))
```

will write the values 2, 6, 11, and 19: all the positions at which "oo" can be found.

To see another example of goal-directed evaluation with generators, consider a famous and difficult problem in parallel processing: given *n* lists, find an entry common to each. (The difficulty this problem gives programmers is one reason why you get so much duplicate junk mail.) A solution in Icon uses the following four lists:

```
l1 := ["Muddle", "Slime", "Eiffel", "WHEREFOR",
       "Intercal"]
l2 := ["Eiffel", "Ickfor", "Intercal", "Prolog"]
l3 := ["Intercal", "WHEREFOR", "Edison", "Slime",
       "Muddle", "Eiffel"]
l4 := ["Prolog", "Slime", "Intercal", "Eiffel"]
```

The Icon expression that will solve this problem (and display the result) is:

```
write (!l1 == !l2 == !l3 == !l4)
```

The bang ('!') operator can be used with any compound structure (lists, strings, tables or csets) to generate its elements, one after the other. The "==" is the string equality operator: it forces the elements of the lists to be compared as though they were strings (which in this case, of course, they are). If you want to display all the names that are common to each list, you can use the *every* keyword again:

```
every write (!l1 == !l2 == !l3 == !l4)
```

The result sequences generators produce are limited to the expression that contains them. If you invoke the same generator at a different point in the program, it starts its result sequence over again from the beginning.

Sometimes it would be nice to get the 'next' result from a generator at various points in the program. To do this, Icon provides co-expressions that bear the same relationship to expressions that co-routines do to functions. You create co-expressions by using the keyword *create*:

```
find_exp := create find ("oo", "Too Soon Too Say GoodBye")
```

Now that the co-expression has been created, you can get a new value at any time like this:

```
value := @find_exp
```

Co-expressions are pervasive in Icon. It may not be obvious, but Icon is always in the process of evaluating some co-expression.

A real world application: *crushc.icn*

One of the most popular uses for Icon is string scanning. One of the first Icon programs I ever wrote used string scanning to remove whitespace and comments from Commodore-Amiga's header files, so that I could fit them all on my floppy disk. To give you a little of the flavor of Icon, and to illustrate the remarkable elegance of Icon's string scanning facilities, I present a more recent incarnation of that program here.

Icon procedures are framed with the keywords *procedure* and *end*, reminiscent of Pascal. As in the C language, program execution always begins in the main() procedure.

Many of the details in *crush*'s main() procedure are not germane to the topic at hand and so won't be discussed in detail. Essentially, it sets up a file stream to read from. If there is a command line argument, it tries to open that file; otherwise, it reads from the standard input. The mesh ('#') character introduces a comment, and comments do not extend past the end of the line.

The first line of real interest for our purposes is:

```
every |read (inputfile) ? write (crush_white ())
```

The question mark is Icon's string scanning operator – the expression before the scanning operator becomes the subject to

be scanned by the expression after the scanning operator. In this case, a line of the file is generated by read(), and set to be the string subject by the scanning operator ('?'). The actual scanning is done by the procedure crush_white(), and the result produced is then output by write().

The crush_white() procedure contains several storage class declarations. The *static* keyword preserves the values of the variables between activations of the procedure. The initial clause executes only once, the first time the procedure is entered, to initialize the variables to their values. This is done for efficiency reasons: since this function will be called many times in the course of the program, it is desirable to only compute these values once.

The variables contain an Icon data type known as a *cset*, simply a set of characters. The twiddle ('~') before each cset literal is the cset complement operation. After the assignment is complete, the variables will contain everything but the characters between the single quote marks. These variables are used during the scan that follows. The result variable accumulates the line sans whitespace that we will return to our caller.

String scanning only needs two matching functions to do its work: tab() and move(). These are called matching functions because they produce actual string values, pieces of the string subject we are scanning.

The values generated by tab() and move() are also the only two values that can change the cursor position during a string scan. They work by remembering the current position of interest, advancing it to a new position, and then producing the string between those two positions. (The global variable *&pos* contains the current position). You can tab() to a specific position in the string, or move() a number of characters forward (or back) from where you are currently.

Armed with our matching functions, let's pause and consider our algorithm for a moment. We want to skip any leading whitespace; and we note that our variable *c_chars* contains every character but a space and a tab:

```
c_chars := ~" \t"
```

So essentially, we want to advance the cursor up to the first character we find that is in the cset *c_chars*. This is done nicely with the Icon expression:

```
tab (upto (c_chars))
```

This expression will fail when it comes to the end of a line (nothing to tab to), so we can also employ it to control the processing of the line by using it as an argument to *while*.

Note that nowhere do we supply the string we are scanning as an argument to upto(); that's the job of the string scanning operator, which sets the default scanning object for these string functions to the one we want.

Now that we have a non-blank, it could be one of four things:

1. A string literal, anything between double quotes
2. A char literal, anything between single quotes
3. A comment, anything between "/*" and "*/"
4. A C operator, identifier, or function

The simplest case is number 4, so let's examine that first. A C operator, identifier, or function can be viewed as a sequence of one or more non-blank characters. So we can again use the characters stored in our variable *c_chars*, but this time with the function many(). Many() is the inverse of the upto() function: it stops at the first character it finds that is *not* in its argument.

```
result := result || tab (many (c_chars))
```

One of SNOBOL's ugliest features was its use of the space character as its string concatenation operator. Icon has fixed that by using double-bars ('||') instead. The line above just concatenates the valid C token produced by tab() to whatever we have stored from previous passes around the loop. The line actually used in the listing is a little different:

```
result ||:= tab (many (c_chars))
```

This line does exactly the same thing as the longer version, but more compactly and efficiently. It is an example of augmented assignment, similar to C's, but which is more pervasive in Icon; all operators except assignment operators themselves can be used in augmented assignment. Here is an example using addition:

```
i := i + 1
```

is the same as

```
i +:= 1
```

The other cases (string and char literals, and comments) all have distinct starting and closing character sequences. To find these sequences, we can use the Icon function match(). Match() is an anchored version of find(): it will only succeed if the string we want to match is at the current position. We use a series of if-else expressions to test each condition:

```
if result ||:= tab (match ("\"")) then
   << code for string literal >>

else if result ||:= tab (match ("'")) then
   << code for char literal >>

else if tab (match ("\*")) then
   << code for comment >>
```

If match() succeeds, tab() produces the string that we then concatenate onto our result string. If it fails, the assignment operation is not performed, and the next *if* clause is evaluated. To throw the comments away, we just tab() past the comment introducer, without concatenating it onto *result*.

As you might guess from the previous code sequence above, the idiom

```
tab(match("string"))
```

is quite common in string scanning. It is so common, in fact, that Icon allows you to abbreviate the sequence by simply placing an equal sign ('=') before the string to match. This is what our program uses:

```
if result ||:= ="\"" then
   << code for string literal >>

else if result ||:= ="'" then
   << code for char literal >>

else if ="\*" then
   << code for comment >>
```

The result is the same, but it's easier to type, and more readable once you get used to it.

The code for handling the string and char literals is quite similar. We want to preserve whitespace inside string and char literals; in fact, we want to preserve everything in literals. We could use *string_chars* (which contains every character but the double quote) as an argument to many(), but we would have to add one to the position produced by the many() function to include the concluding quote. (Remember that many() passes over many characters, stopping before the first character that is not in its argument). We do this often enough to make it worthwhile to define our own procedure, called over(). The code to handle strings, then,

```
result ||:= tab (over (string_chars))
```

just concatenates the string, whitespace, final quote and all, onto the result. The code for char literals is exactly the same, except that we use *char_chars* (which contains everything but a single quote) instead of *string_chars*.

Icon likes to handle strings a line at a time, but C comments can span many lines: clearly, a bit of a problem. Fortunately, a clean, if unorthodox (by Icon standards) solution does exist. The scanning operations store the current string scanning subject in the global variable *&subject*. It is possible to set this global variable directly, although doing so is usually a bad idea. However, in this case, it's just what we need to do the job.

```
while not tab (find ("*/")) do
   (&subject := read (inputfile))
                    | stop (Unexpected EOF!")
      move (2)
```

We use find() here since we want to scan through the entire line for the first occurrence of "*/". If find() fails, then we read() in the next line and store it in *&subject*, to make it the new string scanning object. Icon automatically resets the string cursor (stored in *&pos*) to the start of the string each time you alter *&subject*, so all the string scanning operations will work as expected.

Once we find the end of the comment, we exit the loop, and the new string subject is correctly set for use by all the other pieces of the program. Note that we move the cursor to the right by two characters after exiting the *while* loop, since we have to move past the end comment sequence located by find(). We're almost there; just a couple of things left to do.

Our program now strips out all whitespace, which is hardly correct – whitespace is important as a delimiter in C programs. But we do want to strip any whitespace that is at the end of the line. So we test the position of the cursor like this:

```
if not pos (0) then result ||:= " "
```

The function pos() succeeds if the current cursor position is the same as its argument; it fails otherwise. Icon strings can be indexed starting from the beginning or the end. If you index from the beginning, then you start at one, and advance the index for each character. If you index from the end, then you start at zero, and decrement the index. This is unusual and useful, like most things in Icon. It permits you to use zero to refer to the end of the string, without having to know how long it is. The line above will concat a single blank to our result string if we are not at the end of a line.

The final order of business is the removal of blank lines. This is done by the *return* expression:

```
return ("" ~== result)
```

As mentioned earlier, the "==" symbol is the string equality operator. Preceding it with '~' turns it into the not-equal string operator. If *result* is not equal to the empty string, then the equality test will succeed, and produce *result* as its value. *Return* will then end the procedure and hand this value back to the caller, which will cause it to be displayed with the write() function back in our main procedure. However, if *result* is the null string, the not-equal test will fail, which, as usual in Icon, will cause *return* not to be evaluated. Execution will proceed past *return* and the keyword *fail* will be evaluated. This ends the executing procedure and signals failure to the caller, which causes write() to fail (not be evaluated). No line is written.

That's really all there is to it. Note that we have been able throughout to code at a very high level, close to our intuitive understanding of the problem at hand. Conventional languages require many details, flag variables, loop variables, memory allocations, and other issues that lead programmers into errors because they are not closely related to the problem they are trying to solve.

I am not advocating the overthrow of traditional languages, and their replacement with Icon; languages like C, Modula-2 and assembler can achieve higher performance. But the Icon

program Crushc would conceivably have taken several hours to write in a conventional language – not very practical for a program that, although very useful, might only be run once or twice over a period of years.

Icon on the Amiga

Icon has been available on the Amiga for a few years now, in a version ported by Gary Sarff and myself. It is free, and available on many of the popular information services (Compuserve, BIX, PeopleLink), as well as private BBS systems. I am currently working on a new version that will be much smaller and will provide hooks into the Amiga's operating system for things like menus and windows.

Many Amiga programs are supporting an ARexx interface. It has been suggested by many Icon users that Icon could use an ARexx interface to allow other applications to take advantage of Icon. I think this is interesting, and well worth doing. Perhaps the best approach would be to make a standalone command processor that could call the various pieces of the Icon system as required by the messages it received. This would keep the present clean design of the Icon system, and still allow standalone execution of Icon programs without special shells or command processors.

Listing: *crushc.icn*

```
# crushc.icn - remove whitespace (including comments) from c programs
#
# bugs: Doesn't know about escapes in string's and char literals
#       Assumes end of comment won't occur in string
#
# Copyright (c) 1989 by Scott Ballantyne (sdb)
#
# Please feel free to improve, use and abuse this program.

global inputfile   # input filehandle

procedure main (args)
  local inputfile

  if \args[1] then # provided command line argument
    inputfile := open (args[1]) | stop ("Can't open ", args[1])
  else
    inputfile := &input # standard input

  # now process the file
  every |read (inputfile) ? write (crush_white ())

end

# crushes whitespace in C programs (blanks, tabs and comments) into
# one space.

procedure crush_white ()
  static c_chars, string_chars, char_chars
  local result
  initial
    {
      c_chars := ~' \t' # everything but space and tab
      string_chars := ~'"' # everything but quote
      char_chars := ~'\'' # everything but tick
    }

  result := ""
  while tab (upto (c_chars)) do
    {

      if result ||:= ="\"" then # handle string literals
        result ||:= tab (over (string_chars))

      else if result ||:= ="'" then # handle char literals
        result ||:= tab (over (char_chars))

      else if ="/*" then # handle comments
        {
          while not tab (find ("*/")) do
            (&subject := read (inputfile)) | stop ("Unexpected EOF!")

          move (2) # advance past */ sequence located by find ()
        }
      else result ||:= tab (many (c_chars)) # handle usual stuff

 # if not at end of line, reduce all preceeding whitespace to a single blank
      if not pos (0) then result ||:= " "
    }

    return ("" ~== result)
    fail
end

# return position after last character in chars

procedure over (chars)
  return (many (chars) + 1)
end
```

Background, and additional information

Icon was designed by a team led by Ralph Griswold at the University of Arizona. It is the culmination of 25 years of research into programming language design and non-numerical processing, which he began at Bell Labs when he designed and implemented SNOBOL.

The standard reference on Icon programming is by Griswold and Griswold, and it's called *The Icon Programming Language*, published by Prentice-Hall, ISBN 0-13-449777-5.

For those interested in the techniques involved in implementing modern languages like Icon, you can do no better than *The Implementation of the Icon Programming Language*, published by Princeton University Press, ISBN 0-691-08431-9. This book is also by Griswold and Griswold. Implementing all these tricky new features is not easy, and Griswold has broken much new ground. I consider this book something of a landmark in language implementation techniques and interpreter design.

Seth: Playing The Chaos Game

Surprising order from chaos

by Robert P. Hale and Victor A. Wagner, Jr.
Copyright (C) 1989 Robert P. Hale and Victor A. Wagner, Jr.

Loosely defined, Chaos sits between predictability and randomness. Chaotic phenomena are bounded but can't be simulated. The discovery of simple algorithms that produce incredible complexity is one of the motivating forces behind the study of Chaos.

The opening of the television program, *Nova: The Strange New Science of Chaos*, from WGBH in Boston, goes something like this:

"We're going to play the Chaos game.

1. Pick 3 points in a large triangle and label them. Then pick a random point somewhere on the page.

2. Roll a die, to pick one of the corners, and mark the point halfway from the current point to the corner.

3. Repeat step 2 many times. You would expect to fill the page with lots of dots in a somewhat random pattern. Instead, what you get is this... "

What shows up on the TV screen, almost as if by magic, is a lacy looking pattern made up entirely of triangles of various sizes.

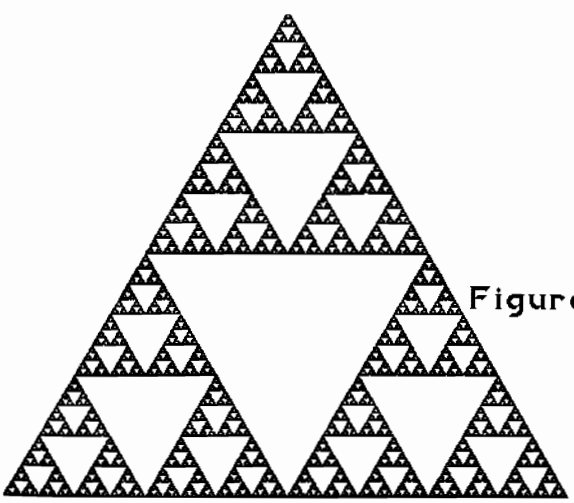


Figure 1

A program and lots of questions

Since this is an Amiga magazine, you can probably guess what ran through our minds as soon as we saw the TV program. The Amiga's built in features, such as simple-to-use graphics and multiple colours helped greatly in our study of this problem. Development proceeded in an experimental "What if we do this?" sort of style. In passing the information along to you, we've attempted to straighten out the organization.

Writing a C program that teaches the Amiga how to play Dr. Barnsley's game is fairly simple:

```
for (i=0; i < ArbitraryLargeNumber; i++)
{
        r = rand() % 3;    /* r = 0, 1, or 2 */
        x = (x + Corner[r].x)/2;
        y = (y + Corner[r].y)/2;
        WritePixel (rp, scale(x), scale(y));
}
```

Sure enough, the multifaceted triangle appears on my screen too.

But that raises more questions than it answers. The first, of course, is why? Not really obvious, not yet anyway. Some other questions present themselves as well:

1. What effect does the placement of the starting dot have?
2. What about colouring the dots to describe where they're going?
3. What is the area of the coloured-in part?
4. What about more sides?
5. What about ratios other than 1/2?
6. Is this a fractal?

Starting point placement

It doesn't seem to matter. If the dot is outside the triangle, it rapidly moves inside it. This makes the procedure an 'attractor' in Chaos terminology.

If the point starts out in an empty area inside the triangle, it rapidly disappears into a full one. In this case, by 'rapidly' we mean less than 10 turns or so. There must be some reason that dots wants to stay out of the blank areas.

Using colour

Since the Amiga is a machine with lots of colours, what can we use colour for? It might be nice to know what corner the dot is heading toward. If we associate a colour (which is easy on an Amiga) with each corner and colour the dot the colour of the corner we're going to, we can check this.

What happens? That's the next surprise. We expected to see the three colours spread salt-and-pepper fashion across the screen. That is most emphatically what does not happen. The grin stays but the cat disappears.

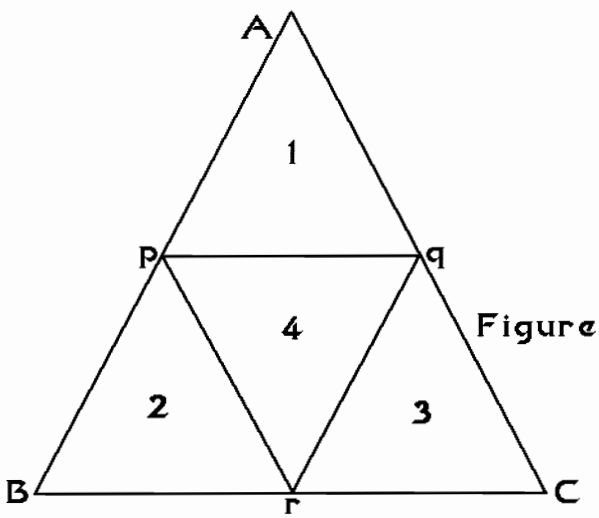


Figure 2

In figure two, the sub-triangle 1 is filled with all of the same colour: colour A. Similarly, 2's pixels are all colour B and 3's colour C.

When you think about it, the distribution of colours makes some sense. Consider that the next dot will be half way towards a corner from the previous. That forces the dot to less than half way from the target corner. Given, for some reason, the middle triangle is unreachable, the Depth = 0 triangles are coloured the same as their corner's. From now on, we'll call this the Depth = 0 display.

History

The Depth = 0 display now seems obvious, at least in retrospect. Where we're going has turned out not to be as colourful as we had hoped. Since we can't predict the future placement of dots, the past is the only thing left. That is, what if we colour the points with the colour of the corner we were heading towards n turns ago? One way to modify the loop to keep track of the history is with the following code:

```
for (i = 0; i < ArbitraryLargeNumber; i++)
{
   r = rand() % 3;
   for (j = MaxHistory-1; j > 0; j--)
      History[j] = History[j-1];  /* shift over dots */
   History[0] = r;
   x = (x + Corner[r].x)/2;
   y = (y + Corner[r].y)/2;
   SetColour(CornerColor(History[Depth]));
   SetPixel(rp, scale(x), scale(y));
}
```

We already know what happens with Depth = 0.

So do we get an explosion in a confectioner's shop for Depth = 1? No. The triangles 1, 2, and 3 in Figure 2 are now each half-sized copies of the Depth = 0 display. The duplication even includes the colours. In each of 1, 2, and 3, colour A is still on the top, B on the left, and C on the right. We end up with 9 distinct single-colour triangles.

Depth = 2? A pattern starts to appear. Each of Depth = 1's single-coloured triangles become miniature Depth = 0 displays. In general, for Depth = n, you'l find 3 to the nth miniature Depth = 0 displays. With a display with only 400 by 400 pixels to play with, these miniature triangles disappear around Depth = 6 or 7.

What this implies is important as well as profound. If you know where a point is, you can retrace the path it took to get there! This follows from the fact that when a dot is in a particular Depth = 0 subtriangle, it must have been headed for that subtriangle's corner. The algorithm is simple. Double the distance from the dot to the nearest corner, and the previous dot is at the point that far away from the corner along the line that passes through the corner and the dot. We can, of course repeat this algorithm further and further back until...

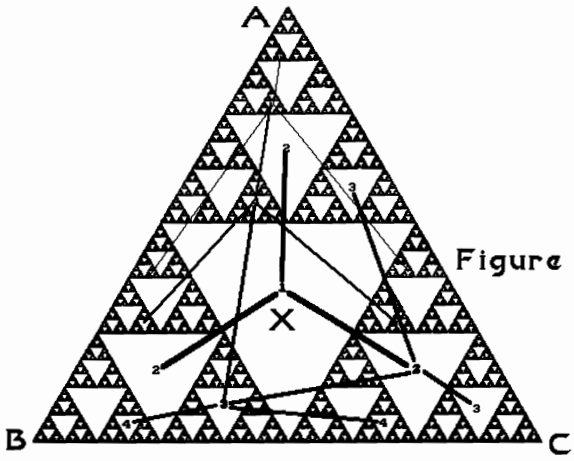
It is worth studying the back-tracking algorithm's action when the point is in the centre triangle (4 in Figure 2). The 'previous' point is outside the triangle. You don't have to have the program to try this!

Another thing to note is that every point in the triangle is half way from some point outside the triangle to *one* of the corners. The backtracking algorithm can't tell where the whole escapade started. The 'path' it generates assumes the previous point was inside the triangle, if possible.

Paths and trees

Let's consider, just for the sake of discussion, starting with a point (marked as 1) in triangle X in Figure 3. We will pretend that we have rolled a die which comes up such that we will succesively head towards corners C, B, A, and A. We have drawn all three possible paths from the 1st point (marked 1) to the 2 triangles labelled 2. From the one which is half way to C, we have drawn 3 possible paths to triangles labelled 3, and so on. If we had drawn all of the possible paths from each of the

possible destinations, the figure 3 would resemble figure 1 attacked by a mad spider. Consider, however, such a set of lines. Each node would have one line coming in, and three going out. Such a collections of lines and nodes is commonly called a tree. It's a tree of degree 3. That is, each node can only exit back to its parent (hence the history that we discovered earlier), and it has 3 children, one for each random number.



Although traversing such a tree in a forward direction requires a die (or some random number generator), traversing it in reverse is simple. Each node has only one parent. We noted earlier that it is *not* necessary for the path inside of triangle to start in any particular area. We can show that any other point at which the path enters the triangle is on a tree which starts in the central empty triangle. We will therefore call trees which *do* start in there Sequoias (hey, we're Californians, and that's what big trees are called here).

How many Sequoias are there? Well, there are an infinite number of points in the centre triangle, and we *know* that no two trees share a point, ergo there are an infinite number of Sequoias.

Empty spaces

The colours showed us the way to further truths. Consider all of the points half way between the line segment AC and the point B (refer to Figure 2). The line formed is pr. Further, consider all of the points along the line segment AB, then all of the points which are half way to B from each of them. The line thusly formed is pB. Finally we find that the line rB is similarly derived from BC. Therefore any points in the entire triangle ABC, for example, which 'head' towards B will end up inside the triangle pBr (better known as triangle 2 in figure 2). Since a triangle is symmetrical, we can argue the same logic for BC towards A and AB towards C. This means that once we're inside one of the Depth = 0 triangles, we can't ever get into pqr. We'll always stay inside the subtriangle we're currently in or hop over pqr.

Note that this is the case even if we came from pqr. As soon we exit pqr for half way towards one of the corners (which happens in one turn), we'll never get back in to pqr. This means that if our back-tracking algorithm generates a point in pqr, we're done. The path must have started there.

If we apply the above logic (looking at the loci of points halfway along our new triangles to each of the corners), we discover that once we leave the large empty space in the middle of triangles 1, 2, or 3, we can never get back.

Area

"I've often seen a cat without a grin", thought Alice; "but a grin without a cat! It's the most curious thing I ever saw in my whole life" - Lewis Carrol, *Alice In Wonderland.*

The area of the coloured in part of the triangle is 0. You can see this in two different ways. The first is to think about having an infinite number of trees. You can easily get mired in 'Aleph Null' and 'Aleph One' infinities.

Happily, there is a simpler method: we simply calculate the area of the empty space. It's clear that the largest empty triangle is 1/4 of the full triangle. If you study it, Depth = 1 empty triangles are 3/16 of the full triangle. A few more feats of triangle counting yield a formula:

$$\lim_{N \to \infty} \frac{1}{4} \sum_{i=0}^{N} (3/4)^i$$

which is, in fact, 1. You can either work out the limit or run a simple program for, say, 10 iterations. It converges very quickly towards 1. Or, if you remember what we were taught in Algebra class:

Consider the series

$$s = r^0 + r^1 + r^2 + \ldots r^{n-1} \qquad (1)$$

if we multiply both sides of the equation by r, we have:

$$r*s = r^1 + r^2 \ldots r^n \qquad (2)$$

if we then subtract (2) from (1) we get:

$$s*(1-r) = r^0 - r^n \qquad (3)$$

Since r^0 is 1 for all r (except ∞) the final equation looks like:

$$S = \frac{1-r^n}{1-r} \qquad (4)$$

in this case we have the limiting case of $n \to \infty$ and since r is less than one, r^∞ is clearly 0. This leaves us with:

$$S = 1/(1-r) \qquad (5)$$

substituting 3/4 for r yields S = 4. Since we want 1/4 of S, the

area in question is 1.

What this says is disturbing. If the empty area inside the triangle is 100 percent, the full area is zero percent! So, there really isn't any shape there after all. The shape is an illusion caused by pixels being significantly fatter than points.

Is it a Fractal and is it Chaos?

Is it a Fractal? A fractal is a shape that has an infinite 'circumference' enclosing a finite area. But what we generate isn't a shape, since its points are not connected. Furthermore, it has no area. So it's not a fractal.

Is it chaotic? Chaotic phenomena aren't as well defined. One of the classic signs of a Chaotic phenomena is dependance in initial conditions, by which the theoreticians mean that if you change the staring points slightly you ought to see radical differences in the final conditions.

Let's consider the game of Chaos with different starting conditions. Mentally pick a couple of points (make them as far apart as you wish). Okay, roll the die. Instead of simply marking the halfway points between our starting points and the selected corner, let us think about this. Draw lines connecting the three points (the two starting points and the selected corner). Now mark the halfway points and draw a line between them also. Figure 2 is a schematic representation of what we have just done.

Suppose we let B and C represent the initial points chosen, and A the corner of the triangle, then p and q would be the halfway points. We notice that we have two similar triangles (ABC, and Apq). Since we know that Ap is one-half of AB, it follows that pq is one-half of BC. Hmmm, after one turn the two points on our different paths are only one-half as far apart. And so it continues. After each 'turn' of the Chaos game we can mentally draw the triangles. We discover yet another pair of similar triangles, and the new points on the paths are only one-half as far apart.

We can continue this game of Chaos, following two separate paths, until the points are arbitrarily close to each other. And once they are close, they will continue to get even closer with each turn. This is *not* sensitivity to initial conditions, this is just the opposite. So it's not chaotic!

So what is it? It's neat, elegant, nonobvious, recursive. Does it have a physical analogy? How about this? It is a random collection of 'above or under water' boolean samples taken n seconds after a stone was thrown in an idealized triangular pond.

Generating regular n-gons

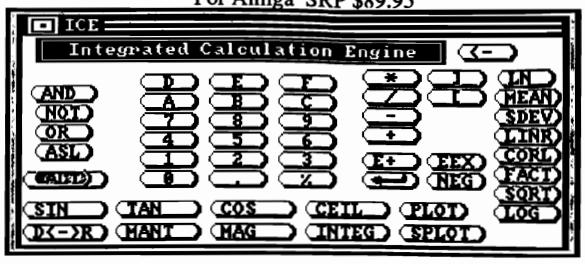
Given our knowlege of triangles, we can pretty well predict other shapes. The problem is, how do we generate the corners for regular n sided shapes (n-gons)?

The algorithm presented here is based on the fact that in a regular (equal sides) n-gon, all of the points lie on a circle.

continued...

I.C.E.

Integrated Calculation Engine
For Amiga SRP $89.95



Who needs it?

Students Programmers Scientists Teachers Financiers Statisticians Businessmen Accountants Researchers Technicians Realtors Leasors Rental agents Contractors Engineers Mathematicians Numbercrunchers Everywhere!

Why?

I.C.E. makes number work easy! I.C.E. is a programmable calculator construction set. Create calculators designed to do the jobs you need done. Using the mouse, select the buttons (functions) you want for your customized calculators. Add the graphic plot &print functions, choose the layout and number system you find comfortable, bind macro programs to a single key for truly powerful versatility. Then use your I.C.E. creations with other programs like spreadsheets and databases. I.C.E. is not copy protected and multitasks beautifully! Now you can have programmable calculators with the features you choose. I.C.E. calculators include an integral program editor which allows you to create complex formulae from over fifty available functions.

KING PUBLISHING

921-32nd Street Sioux City IA 51104 Phone 712-252-4604

The Disk Mechanic

Give your disks a Tune-up

The Disk Mechanic is a comprehensive collection of utilities for Amiga DOS, including TuneUp, DoubleBack, DiskRepair, and WorkShop.

TuneUp: With TuneUp you can improve hard disk or floppy disk performance by up to 400%. Not a caching program, TuneUp is a sophisticated disk optimizer that actually re-organizes the information on your disk to improve system efficiency. TuneUp is fully compatible with Amiga DOS.

DoubleBack: DoubleBack is a high speed archival hard disk back-up program. DoubleBack supports multiple source and destination drives, file selection by name, date, protection, directory, and file note. Back up a full 20 megabytes in under 40 minutes. Selectively copy files between hard disks and partitions. Protect your data!

Only $89.95

Sample TuneUp benchmark of some standard 3.5" disks:

Volume Name	TuneUp	BEFORE	AFTER	IMPROVED
Workbench 1.3	3:30	4.86	2.70	79%
Digi!	3:05	9.44	5.22	80%
WordPerfect	4:42	12.36	5.37	130%
SuperBase	3:37	19.39	9.66	100%
data	1:43	14.30	4.68	205%
archives	3:40	28.43	9.52	299%

* Times given are the average of three runs on a standard A2000 internal floppy drive. The TuneUp time is the time required to process the disk (in minutes and seconds). The BEFORE and AFTER times are the time (in seconds) required to get a directory of the disk. The more use a disk has had, the better the improvement!

Demo disk Available; call for details.

DiskRepair: With DiskRepair it is possible to recover accidently deleted or discarded files, salvage files from damaged disks, and reconstruct a corrupted hard disk. DiskRepair works with hard and floppy disks, and is compatible with the Fast Filing System.

WorkShop: For the expert user, the disk WorkShop is a full featured disk file and block editor. Examine and modify disk files, search for viruses, and change menus or requesters in popular programs.

The Disk Mechanic also includes a dozen small CLI based utilities to make managing your disks easier. We are now shipping version 2.5 with full support for Amiga DOS 1.3 and the Fast Filing System. The Disk Mechanic is not copy protected.

Dealer inquiries welcome
Amiga is a trademark of Commodore Amiga Inc.
WordPerfect is a trademark of WordPerfect Corp.
Digi! is a trademark of Aegis Development
SuperBase is a trademark of Precision Software, Ltd.

Lake Forest Logic Inc.
28101 E Ballard Road
Lake Forest, IL 60045
(312) 816-6666

MasterCard VISA

The steps then, are:

1. Find the radius of the circle.
2. Determine the starting point.
3. Iterate around the circle.

We need to determine the starting point so we can force the algorithm to generate corners of the n-gon so that there is an edge on the bottom.

If the number of sides is odd, r = WindowHeight/(1+cos(pi/i)). If the number of sides is even, r = (WindowHeight/2) / (cos(pi/i)). The base coordinates also depend on whether the number of sides is odd:

```
if (NumSides & 1 == 1) { /* odd? */
    r = WindowHeight/(1+cos(pi/NumSides));
    cx = WindowHeight/2;
    cy = r;
}
else {
    r = (WindowHeight/2)/(cos(pi/i));
    cx = WindowHeight/2;
    cy = WindowHeight/2;
}
```

We now need to generate the points themselves:

```
theta = (2*pi)/NumSides;
for (point = 0; point < NumSides; point++) {
    Corner[point].x = r * sin(theta*((2*point)/2)+cx);
    Corner[point].y = r * cos(theta*((2*point)/2)+cy);
}
```

More sides

The program gives the user a choice of n-gons with from 3 to 15 sides. This way, a simple 16-colour Amiga window can display the history using colour, and we still have a background colour.

With more sides, however, we generally see a 'salt and pepper' effect. This is due to our continued use of half the distance from the last point to the next chosen. If we add the ability to change this ratio, say by allowing the user to provide the denominator and numerator, we change the algorithm as follows:

```
r = rand() % NumSides;
/* record history as above */
x = (denom*x)/num+(num*Corner[r].x)/denom;
y = (denom*y)/num+(num*Corner[r].y)/denom;
```

which means that the new coordinates are num/denom of the way from the old coordinates to the chosen corner.

Our personal favourites have NumSides of 7, 11, 12, 13 and ratios that cause the 'rings' to overlap slightly.

Using rand()

n the code examples so far, we've used the standard C rand() function. If, however, we use this with NumSides a power of two, we get a very short repeat cycle, and hence very few dots. Lattice provides lrand48(), which doesn't solve this problem either. We had to finally use lrand48() and divide by a constant before doing the '% NumSides'. Perhaps one use of the algorithm is as yet another random number generator test.

Floating point or integer?

The program uses (standard Lattice 32 bit) integers to store the points, and for most everything else. The integers are scaled by a factor of 10000, which is to say that we must divide a coordinate by 10000 to get actual pixel locations. Viewing it differently, we keep track of the points as if we were drawing on a 4,000,000 by 4,000,000 screen. When it comes time to draw it on a 400 by 400 screen, we divide by 10000.

While you might think that this is to increase speed, we did it for another reason. Stated simply, integer (or really, fixed point) is more accurate in this case. In the same number of bits, floating point must cover much more 'distance', +/- 10^{308} in the case of Lattice. While it is true that many of the points between 0 and 399 (the maximum screen height supported in the program) are contained in the normal floating point scheme, we get many more, and at a very predictable difference (1/10000 apart) with fixed point. Oh yes, it *is* faster as well.

The program

The program is called Seth. Since all of our graphs fit nicely in a square, we used a 400 by 400 window. Seth uses a fixed palette of 15 colours plus a background colour. That limits the number of sides to 15, but if you can't see the pattern by then, a few more sides isn't going to help.

The program uses menus and gadgets to allow the user to control the number of sides, history depth, background colour, and divisor ratio. The program continues to play the game until told to do something new. We don't, in fact, have *ArbitraryLargeNumber* as a loop control constant.

The program is very computationally intensive. The only I/O it does for long stretches is to the screen. If you want to run other tasks in the machine as well, you should invoke this program at a low priority.

Program availability

The program, is of course, on the disk that goes with this issue. It was written in Lattice C (5.0). It is called Seth because Seth is the Egyptian God of Chaos. From what we've seen however, Horus, God of Order, might be more appropriate.

A CSA TURBO 68020 FOR ONLY $399?

Hard to believe isn't it? CSA broke the price barrier in 32 bit technology. Now you don't have to settle for a far less capable 68000 accelerator, you *can* have affordability, capability, and speed in one easily installed package.

"1000% faster than my stock Amiga™?"

CSA's new 68020 *Midger Racer*™ flies through MaxiPlan™ spreadsheets over 60% faster than the standard Amiga™, and over 50% faster than 68000 based accelerators. With the addition of a 12 MHz 68881 co-processor, the *Midget Racer*™ can crunch 2 Million FLOPS over 1000% faster than a stock Amiga™.

"600% faster than a 68000 accelerator?"

In fact, programs like Sculpt & Animate 3D or 4D™, Draw 2000™, and X-CAD™ have been written to directly access the 68020 & 68881, and may not even *run* with a 68000 based accelerator. And with a 12 MHz 68881, CSA's *Midget Racer*™ is up to 600% faster than a 68000/68881!

"All that for only $399?"

No, there's more! It's also easily installed into ***any*** Amiga, supports an '881/2 up to 33 MHz, and it's available now. Don't settle for second best, call your Amiga dealer or CSA today!

CSA. We set the standards in performance.

CSA

Computer System Associates, Inc.
7564 Trade Street
San Diego, CA 92121
619-566-3911 FAX: 619-566-0581

Trademarks: Amiga: Commodore-Amiga, Inc.; MaxiPlan: Oxxi, Inc.; 68000/020/881/882: Motorola, Inc.; Sculpt & Animate 3D/4D: Byte by Byte, Inc.; Draw 2000: Aegis, Inc.; X-CAD: Taurus, Inc.; Midget Racer: Computer System Associates, Inc.

COLOR RIBBONS & PAPER

RIBBONS: Red, Blue, Grn., Brwn., Purple, Yel.

Ribbons Price Each	Black	Color	Heat Transfer
Brother M1109	4.95	5.95	7.00
C. Itoh Prowriter Jr.	7.00	9.00	–
Citizen 120D/180D	5.00	6.00	7.95
Commodore MPS 801	4.50	5.25	5.75
- MPS 802/1526	6.25	7.25	–
- MPS 803	4.95	5.95	7.00
- MPS 1000	3.95	4.95	6.75
- MPS 1200/1250	5.00	6.00	7.95
- 1525	6.50	8.00	–
Epson MX80/LX800	3.75	4.25	6.75
Okidata 82/92	1.75	2.25	4.50
Okidata 182/192	6.50	7.50	6.00
Panasonic K-XP 1080	6.75	7.75	–
Seikosha SP 800/1000	5.25	6.50	7.95
Star SG10	1.75	2.25	4.50
Star NX10/NL10	5.00	6.00	7.95
Star NX1000	4.50	5.50	6.75
Star NX1000C – 4C	–	8.75	–

T-SHIRT RIBBONS (Heat Transfer) – Red, Blue, Grn., Brwn., Purple, Yel., Blk. Call For Price & Availability.

COLOR PAPER

BRIGHT PACK–200 Sheets/50 each color: Red, Blue, Green, Yellow. 9 1/2 x 11 – $11.90/pk.

PASTEL PACK–200 Sheets/50 each color: Pink, Yellow, Blue, Ivory. 9 1/2 x 11 – $11.90/pk.

COLOR BANNER BAND PAPER – 45 ft./roll–$9.95/ea.

For ribbons & paper not listed above, call for price & avail. Price & spec. subject to change w/o notice. Min. order $25.00. Min. S&H $3.50 min. Visa, MC, COD.

RAMCO COMPUTER SUPPLIES
P.O. Box 475, Manteno, IL 60950 U.S.A.
(US) 800-522-6922 • (Canada) 800-621-5444
815-468-8081

AMIGA COMMODORE CHIPS

6526	$12.95	8364 (PAULA)	$56.95
6567	$16.95	68010-8	$29.95
6510	$10.95	8362 (DENISE)	$56.95
6581	$11.95	8370 (F.Agnus)	$61.50
PLA	$12.95	8520-A1	$17.95
901 series ROMS	$11.95	8386 (GARY)	$17.25
AND MANY OTHERS		A501 upgrade (A500)	$147.00

COMMODORE DIAGNOSTICIAN II

JUST OUT — a newly revised/updated version of the COMMODORE DIAGNOSTICIAN which sold over 10,000 copies world-wide. The COMMODORE DIAGNOSTICIAN II locates faulty chips on all COMMODORE computers/1541 drives and has different sections such as "cross reference #s". This diagnostic tool had a fantastic full page review in last years "Computer Shopper Magazine". Cost is $6.95 prepaid to North America.

We also sell RAM/SIMMS testers, diagnostic evaluators, cables and other exclusive new products for Commodore and IBM.

Send for catalog on exclusive new products

The Grapevine Group, Inc.
35 Charlotte Drive
Wesley Hills, N.Y. 10977
1-800-292-7445 FAX 914-354-6696
914-354-4448

MasterCard

Dealer Prices Available
Prices subject to change

VISA

Computer Programmers

Gold Disk Inc. of Mississauga is looking for two C programmers to enhance their professional desktop publishing and illustration system software on the AMIGA and IBM PC.

Candidates should have a minimum of two years experience working with C and have experience working with a major project.

Assets include: AMIGA and IBM PC experience, knowledge of 68000 assembly and experience on desktop publishing and illustration software.

Salary negotiable based on experience.

Contact Henry Gin at (416) 828-0913 between 9 a.m. and 11 a.m. (EDT)

Haitex X-Specs 3D

3D glasses from Haitex Resources
Suggested retail price: $124.95 US

Review by Paul Bosacki

Coming from a generation more familiar with cardboard glasses with red and green filters, I was intrigued when the X-Specs package came in. True 3D is a pretty hefty claim. So first, try on the glasses: large black affairs, that didn't look as though they would be very comfortable. I was wrong – in fact, even eyeglass wearers will find them comfortable. The interface plugged directly into the joystick port, and the X-Spec glasses plugged into the interface with a subminiature phono-jack. Another set of glasses can be connected to the interface.

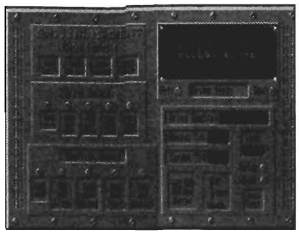
Time to examine the software. *Space Spuds* proved to be a comic shoot'em-up with a space theme. Seems you go after large, interstellar potatoes and french fries with such incredible weapons as the 'Lippo Ray' and the 'Fat Blaster.' The visual effect was amazing. The screen became an infinitely deep window into space. Out of the vast distance (and it really did seem vast), came the adversary. I was impressed.

Next in were 3D molecular models which seemed to rotate in the space before the monitor. *Cubes Demo* brought slowly turning, wire-frame cubes to rest about 12 inches in front of the monitor. *Make3D* allows the creation of 3D images using IFF format pictures. This is best achieved with packages such as *Sculpt-3D* or *Turbo Silver.* A scene is "shot" from two different angles, then the two images are loaded into *Make3D.* Offsetting the left and right image creates the 3D effect.

X-Specs 3D are, in the words of the developer Haitex Resources, a "stereoscopic vision system". The human brain processes steroscopic images from two flat images viewed from slightly different angles. X-Specs control what each eye sees by closing and opening an optical shutter at 60 frames per second. On even numbered openings, an image is displayed to one eye while on odd numbered openings, a somewhat offset image is displayed to the other. The optical shutter employs high-speed liquid crystal technology.

When the images are offset slightly, the human brain is 'tricked' into processing a three-dimensional image. Depending on the angle of separation between the two images, the three dimensional effect can be either enhanced or minimized. This is where software like *Make3D* is handy.

Although the supplied demos are a bit whimsical, Haitex claims some very interesting applications for their hardware: from the medical to the scientific. In fact, the molecular modelling demo comes courtesy of the University of Cinncinati's Department of Chemistry. Apparently, over 40 independent software development firms are working on X-Specs 3D compatible software.



Ami...Alignment System

Floppy disk diagnostic software
from Free Spirit Software
Suggested retail price: $39.95 US

Review by Doug Lee

Ami...Alignment System (AAS) purports to automate testing of several facets of floppy disk drive performance: track alignment, drive motor speed, track integrity, and read/write data transfer speeds. It also features a mode that provides manual control of the motor and read/write head.

AAS has some real polish: a slick, intuitive user interface; an intelligible, informative manual; and smooth performance in its evaluations. To test its reliability, technician Doug Lee compared the values generated by AAS with those produced with an oscilloscope and a professional drive alignment disk (which costs about $300 Can.). His report follows:

"This program is useful for stepping the head to a particular track when doing manual alignments with a oscilloscope. It is also useful for measuring disk speed – the software probably does this by counting the index hole pulses generated by the Amiga hardware.

"The AAS, like other Amiga alignment software, can only read the digital signals output from the drive connector – the analog drive signals needed for truly accurate alignment are not available to the computer. AAS can tell you if a given track is close enough to spec to be properly read by the CPU – a pass/fail mode of operation. The program's degree of accuracy depends on whether the AAS alignment disk was recorded on a drive with extra narrow tracks and reduced recording current or on a standard drive.

"I have done some tests with this program usingan oscilloscope and a professional analog alignment disk. I have discovered that on a defective drive that runs too slowly, the AAS reveals the speed problem without difficulty. However, because of the speed problem, the defective drive can't read the data from the program's alignment disk. Since it is an AmigaDOS disk, *any* drive fault will render it unreadable, unlike the oscilloscope method, which allows you to observe analog data directly off the disk.

"Using the oscilloscope and professional alignment disk, I found the radial track alignment of my defective drive to be closer than the resolution of my scope. On the other hand, my working df0: drive, which the AAS said was fine, was noticeably out of alignment when checked with the oscilloscope. It actually isn't out far enough to cause reading problems, but any disks written on this drive won't be 100% accurate."

Doug's results suggest that AAS is a worthwhile product for such tasks as drive speed checks, media checks and head cleaning. But use its alignment capabilities with caution.

News Link

Motorola announces two new 68xxx's: Motorola announce the availability of a 50 MHz version of its 68030 microprocessor. Apparently, the new microprocessor offers the fastest clock in the industry. The chip performs at 12 MIP, double that, Motorola claims, of all other conventional processors currently available.

The 50 MHz '030 is software compatible with all microprocessors in the 68000 family, The new '030 entered general sampling in May. Volume production will probably have begun by the time you read this. The part is priced at $650 US for sample quantities.

Also Motorola has disclosed the features of its new 68040, a virtual memory 32-bit MPU that remains software compatible with other members of the 68000 family. On a single chip, the MC68040 incorporates a '030 compatible integer processing unit, an IEEE-compatible floating point unit, fully independent instruction and data caches, and a page memory mangement unit. Don't go looking for it yet though. Motorola released a product preview only. Contact: Motorola Microprocessor Products Group, 6501 William Cannon Drive W., Austin, TX 78735-8598, USA.

V1.3 of ARP released: Microsmiths, Inc. has announced the V1.3 release of ARP, the AmigaDOS Resource Project. ARP is a cooperative effort by a group of Amiga developers to enhance the basic operating environment on the Amiga. The V1.3 release of ARP includes the arp.library, a shared system resourcer for programmers, and the ARP CLI commands which are compatible enhancements for the AmigaDOS V1.3 CLI commands.

V1.3 includes many enhancements to both the commands and arp.library, and is compatible with Commodore's *V1.3 Enhancer* command programs and documentation. Among the enhancements are an ARP Shell, and improvements to the standardized file requester in the arp.library. The most significant enhancement is the introduction of the ARP install program. Microsmiths feels certain that their installation program will set a new industry standard for ease of use of installation utilities.

The V1.3 ARP release disk will be available soon from ARP Support (*probably by the time you read this -Ed*). Included will be the updated user and programmer documentation, support materials, and the updated CLI commands and library. Anyone interested can send a self-addressed stamped envelope, and they will be notified as soon as diskette mailing begins. Contact: Arp Support, c/o Microsmiths, Inc., PO Box 561, Cambridge, MA 02140, USA.

ST-506 hard drive adaptor for the Amiga 2000: Spirit Technology has developed a low-cost alternative to the SCSI hard drive. Off-the-shelf ST-506 hard drives can now be adapted to A2000s with Spirit's *HDA-506-2 Hard Drive Adaptor.* The adaptor is a ZORRO BUS version of Spirit's HDA-506 Hard Drive Adaptor currently in use with IBM type drives on Amiga 500s and 1000s. It can be plugged into any ZORRO slot in the A2000 and strapped to a peripheral drive bay to control any 3.5 inch or 5.25 inch ST-506 hard Drive. Additionally, an optional steel rail is available to convert it to a hard card configuration for internal installation in the A2000.

When installed as a hard card, the 3.5 inch ST-506 drive can be mounted on either side of the rail. This permits more installation choices by allowing non-users of the Bridge Board to plug the system into the last ZORRO socket where the drive will fill the space of the two 32-pin sockets normally occupied by the Bridge Board.

Spirit claims HDA-506 speeds are comparable to, and often exceed, the speeds of more expensive 8-bit SCSI systems. Read speeds in excess of 300KB per second have apparently been acheived using the AmigaDOS 1.3 Fast File System, an OMTI-5520 ST-506 controller and most any typical ST-506 hard drive. No price available at press time. Contact: Spirit Technology Corp., 220 West 2950 South, Salt Lake City UT, 84115, USA.

Amiga gets new LOGO: Commodore Business Machines, Inc. announced the launch of a new educational package for the Amiga - AMIGA LOGO. It has many operational features that distinguish it from other LOGOs. Some of the features include pull-down menus and the use of the mouse for free-hand drawing, as well as support for the Amiga's extensive graphics capabilities. The Amiga's built-in speech capabilities allow for help screens that recite the instructions to the pupils.

AMIGA LOGO was developed by Carl Sassenrath, one of the orginal designers of Amiga's multi-tasking operating system. AMIGA LOGO runs on all Commodore Amiga computers with a minimum of 512K memory and a single disk drive. AMIGA LOGO will be available in July. Contact: Higgins Public Relations, 338 Newbury St., Boston MA, 02115, USA.

Communications library for C: Lattice introduces the new *Lattice Communications Library*, a set of high-level and low-level C functions for creating applications that perform asynchronus communications.

Functions include the ability to select, open and close communication port; set baud rate, parity, data bits and buffersize. Other functions dial the phone, reset the modem, send and

receive file using ASCII, XMODEM, YMODEM or KERMIT error-checking protocols. All functions have been designed to produce fast, efficient programs, and each routine has been thoroughly tested and debugged.

The library is $250.00 US with source code available for another $250.00 US. Contact: Lattice, Inc., 2500 South Highland Ave., Lombard, IL 60148, USA.

ImageLink goes into pre-release: Active Circuit, Inc. has announced the shipping of pre-release versions of *ImageLink*, a dynamic image conversion and previewing system for the Amiga. The version shipping is sent with a photo-copied manual, not typeset. However, those who purchase at this time, will have the release documentation subsequently shipped at no additional charge.

ImageLink allows the Amiga to be turned into an image conversion workstation. The basic program includes file conversion modules for IFF, *Sculpt and Direct*, *DigiView*, Macintosh, *Targa File* and *Turbo Silver*. Additional modules are available, and custom modules by arrangement. ImageLink is sold at $299.95 US. Contact: Active Ciruit, Inc., 106 Highway 71, Suite 101, Manasquan, NJ 08736, USA.

Animation from Aegis: Aegis is shipping *ANIMagic*, their animation editing and special effects software. ANIMagic works with most popular paint and animation programs, performing special effects with ANIM* format animations or IFF picture files (including HAM mode). Digital video effects like spins, page turns, venetian-blinds, strobes, and unusual colour and tiling effects are possible. Using the ANIM editor, the resulting animations can be edited, enhanced or spliced together for longer movies. ANIMagic is intended for any application where a combination of graphics and special animation effects are required. ANIMagic retails for $99.95 US. Contact: Aegis Development Inc., 2115 Pico Blvd., Santa Monica, CA 90405, USA.

A colour matching system for the Amiga: *Palette Printer* is a colour match system for the Computer Graphic Design Field. The utility contains files to print out the colour range of any in-house printer. Designers can compare their Palette Printer colour charts to any colour charts already used to select colours for colour printing. By comparing the charts, users can select colour settings in any Amiga paint or desktop/electronic publishing programs to create matching colour.

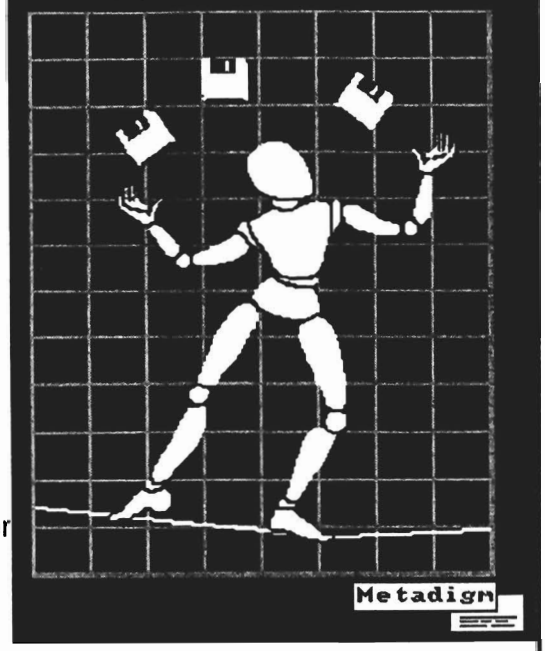
Dots-per-inch as low as 86 by 86 will yield a large array of saturated colours from over 850 colour samples Palette Printer will produce. Higher dpi capabilities will create more usable colours. As higher resolution printers develop and become more affordable, Palette Printer will produce more colours.

Palette Printer colour charts are designed to be printed by all Amiga paint programs. The suggested retail price is $29.95 US. Contact: Ontological Survey, PO Box 17488, Milwaukee WI, 53217, USA.

NEW Release ver 1.10

saf-T-net

- Full Intuition based Interface
- Selective Restore
- Amiga wildcards
- Multiple indexing for greater **safety**
- Over 907K per floppy



Hard Disk Backup

for your Amiga

available at your dealers **now**

or from (714) 955-2555

Metadigm, Inc. 19762 MacArthur Blvd. Suite 300; Irvine CA 92715

GAME PROGRAMS WANTED!

Attention Developers and Freelance Programmers!

Virtual Reality Laboratories, Inc. is looking for a few sharp new entertainment game and educational game programs for the Amiga. To be accepted, these programs must be highly graphic with excellent sound capability. For a rapid review of your work send for guidelines to VRLI, Box 609, Atwood, CA 92601.

Do not submit until you have received and read our guidelines, but please do send for the guidelines TODAY!

Reader Service Card

Fill out this card (please print) or attach your peel-off mailing label, and circle the reader service number for the ads that you'd like more information about (see Advertiser Index below).

Name _____

Company _____

Address _____

_____

Postal/Zip Code _____

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

TA V2I6

Advertiser	Page	Reader Service #
ASDG Inc.	45	1
Black Belt Systems	13	2
Computer Solutions	27	3
CSA (Computer Systems Associates)	77	4
Delphi Noetic Systems Inc.	45	5
Delta Research	27	6
Electronics 2001	39	7
Fishers Computers & Software	39	8
Free Spirit Software Inc.	39	9
Gold Disk	77	10
Intercomputing Inc.	1	11
King Publishing	75	12
Lake Forest Logic Inc.	75	13
M.A.S.T.	5	14
Metadigm Inc.	80	15
Microsmiths Inc.	21	16
Microway	Outside Back Cover	17
Newtek Inc.	11	18
R & DL Productions	11	19
Ramco Computer Supplies	77	20
The Grapevine Group	77	21
The Puzzle Factory	13	22
The Software Shop	41	23
Vidtech International Inc.	Inside Front Cover	24
Virtual Reality Laboratories	80	25

for the AMIGA Transactor

The Magazine for Amiga Programmers

Please print your name and address below (or transfer your mailing label)

Check here for change of address ☐

Name: _____

Address: _____

City: _____ *Province/State:* _____

Zip/Postal: _____ *Phone:* _____ *CompuServe ID:* _____

NOTE: Prepayment required. Purchase orders will be accepted ONLY if accompanied by payment.

Subscribers only: please enter subscription number from mailing label ☐

Subscription Order Section

PLEASE NOTE THAT TRANSACTOR FOR THE AMIGA IS NOW PUBLISHED MONTHLY

	Canada ($Cdn.)			U.S.A. ($US)			Foreign ($US)			Air Mail ($US, overseas only)		
	Mag	Disk*	Both*	Mag	Disk	Both	Mag	Disk	Both	Mag	Disk	Both
12 issues	☐35.00	☐123.00	☐158.00	☐27.50	☐101.00	☐128.50	☐38.50	☐119.50	☐157.50	☐ 73.00	☐ 145.00	☐ 218.00
24 issues	☐65.00	☐220.00	☐270.00	☐49.00	☐160.00	☐190.00	☐69.00	☐220.00	☐290.00	☐ 135.00	☐ 250.00	☐ 390.00

* Ontario residents please add 8% of disk subscription price for provincial sales tax. No PST on magazine subscriptions.

Combo Subscription Bonus!

Subscribe to BOTH the magazine and the disk and receive a FREE extra issue on your subscription!

Product Order Section

Ontario residents please add 8% PST on all items

$Cdn.	$U.S.	
☐ 14.95	12.95	Transactor Classic Amiga Disk #1
☐ 14.95	12.95	Transactor Classic Amiga Disk #2

Fish Disks $6.95 US, $7.95 Cdn. each

Note: Disk numbers not yet released may be ordered in advance - simply add numbers to end of list. Disks 57, 80 and 88 are not available.

☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7	☐ 8	☐ 9	☐ 10	☐ 11	☐ 12	☐ 13	☐ 14
☐ 15	☐ 16	☐ 17	☐ 18	☐ 19	☐ 20	☐ 21	☐ 22	☐ 23	☐ 24	☐ 25	☐ 26	☐ 27	☐ 28
☐ 29	☐ 30	☐ 31	☐ 32	☐ 33	☐ 34	☐ 35	☐ 36	☐ 37	☐ 38	☐ 39	☐ 40	☐ 41	☐ 42
☐ 43	☐ 44	☐ 45	☐ 46	☐ 47	☐ 48	☐ 49	☐ 50	☐ 51	☐ 52	☐ 53	☐ 54	☐ 55	☐ 56
☐ 58	☐ 59	☐ 60	☐ 61	☐ 62	☐ 63	☐ 64	☐ 65	☐ 66	☐ 67	☐ 68	☐ 69	☐ 70	☐ 71
☐ 72	☐ 73	☐ 74	☐ 75	☐ 76	☐ 77	☐ 78	☐ 79	☐ 81	☐ 82	☐ 83	☐ 84	☐ 85	☐ 86
☐ 87	☐ 89	☐ 90	☐ 91	☐ 92	☐ 93	☐ 94	☐ 95	☐ 96	☐ 97	☐ 98	☐ 99	☐ 100	☐ 101
☐ 102	☐ 103	☐ 104	☐ 105	☐ 106	☐ 107	☐ 108	☐ 109	☐ 110	☐ 111	☐ 112	☐ 113	☐ 114	☐ 115
☐ 116	☐ 117	☐ 118	☐ 119	☐ 120	☐ 121	☐ 122	☐ 123	☐ 124	☐ 125	☐ 126	☐ 127	☐ 128	☐ 129
☐ 130	☐ 131	☐ 133	☐ 134	☐ 135	☐ 136	☐ 137	☐ 138	☐ 139	☐ 140	☐ 141	☐ 142	☐ 143	☐ 144
☐ 145	☐ 146	☐ 147	☐ 148	☐ 149	☐ 150	☐ 151	☐ 152	☐ 153	☐ 154	☐ 155	☐ 156	☐ 157	☐ 158
☐ 159	☐ 160	☐ 161	☐ 162	☐ 163	☐ 164	☐ 165	☐ 166	☐ 167	☐ 168	☐ 169	☐ 170	☐ 171	☐ 172

The Transactor for the *AMIGA* Disk $12.95 US, $14.95 Cdn. each

(these disks correspond to issues of **Transactor for the *AMIGA***)

☐ Vol. 1, Iss. 1 ☐ Vol. 1, Iss. 2 ☐ Vol. 1, Iss. 3 ☐ Vol. 1, Iss. 4 ☐ Vol. 1, Iss. 5 ☐ Vol. 1, Iss. 6

☐ Vol. 2, Iss. 1 ☐ Vol. 2, Iss. 2 ☐ Vol. 2, Iss. 3 ☐ Vol. 2, Iss. 4 ☐ Vol. 2, Iss. 5 ☐ Vol. 2, Iss. 6

Add flat $2.00 shipping & handling for any number of items ordered from above box.

Back Issues: $4.50 U.S., $4.50 Cdn. (filled box=SOLD OUT)

☐ Vol. 1, Iss. 1 ■ Vol. 1, Iss. 2 ☐ Vol. 1, Iss. 3 ☐ Vol. 1, Iss. 4 ☐ Vol. 1, Iss. 5 ☐ Vol. 1, Iss. 6

☐ Vol. 2, Iss. 1 ☐ Vol. 2, Iss. 2 ☐ Vol. 2, Iss. 3 ☐ Vol. 2, Iss. 4 ☐ Vol. 2, Iss. 5

Ontario residents please add 8% provincial sales tax for items ordered from this box.

☐ Visa ☐ MasterCard ☐ Cheque/MO enclosed Cheque #_____ Dated ____/____/____ Amount _____

Acct. # _____ Expires ____/____ Signature _____

Please allow 6-8 weeks for delivery in the U.S., 4-6 weeks in Canada.

FOLD ALONG SCORES WITH CORRECT REPLY CARD ON OUTSIDE. USE TAPE, PLEASE - NO STAPLES OR GLUE.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 390 BUFFALO, NY
POSTAGE WILL BE PAID BY ADDRESSEE

Transactor
P.O. Box 338 Station 'C'
Buffalo, NY 14209-9990

place stamp here

Transactor
10-85 West Wilmot Street
Richmond Hill, Ontario, Canada
L4B 1K7

Advertising in Transactor for the Amiga

Interested in advertising in *TransAmi?* Call Mike Grantham at 416/764-5274

Business Reply Mail
No Postage Stamp Necessary
if mailed in Canada

Postage will be paid by:

Canada Post Postes Canada
46487

Transactor
501 Alden Road
P.O. Box 3250
Markham Industrial Park
Markham, Ontario
L3R 9Z9

AMIKIT IS HERE!

AMIKIT IS HERE!
AMIKIT IS HERE!
AMIKIT IS HERE!
AMIKIT IS HERE!
AMIKIT IS HERE!

No more frustrated hours spent staring at your computer screen! No more blundering through the Workbench or CLI -- the ultimate aid for novices and experts alike has arrived!

Step by step, AmiKit helps you explore and benefit from the awesome power of your Amiga. From the most basic to complex commands, AmiKit teaches you how to gain control and confidence using your machine, whether its an A500, 1000 or 2000. And what's more, AmiKit includes over thirty Public Domain software programs, including: Workbench 1.3; a virus checker; a database; a spreadsheet; a word processor; games; and much more!

AmiKit is one of the most effective ways sucessfully to navigate your way throughout the workbench and CLI.
As a special introductory offer from Transactor for the Amiga, you can get AmiKit for $34.95* (plus $2.50 for shipping) -- $5.00 off the recommended retail price -- an amazing bargain! When Amiga User International, the leading European magazine, reviewed AmiKit, they were extremely pleased with the package. We know you will be, too!

To: Transactor For the Amiga
10-85 West Wilmot St.
Richmond Hill, Ontario, CANADA
L4B 1K7
I would like to purchase AmiKit at the special price of $34.95 + $2.50 shipping ($37.45)*.
Please make cheques payable to Croftward Publishing, Inc., or debit my Mastercard/Visa account:

☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

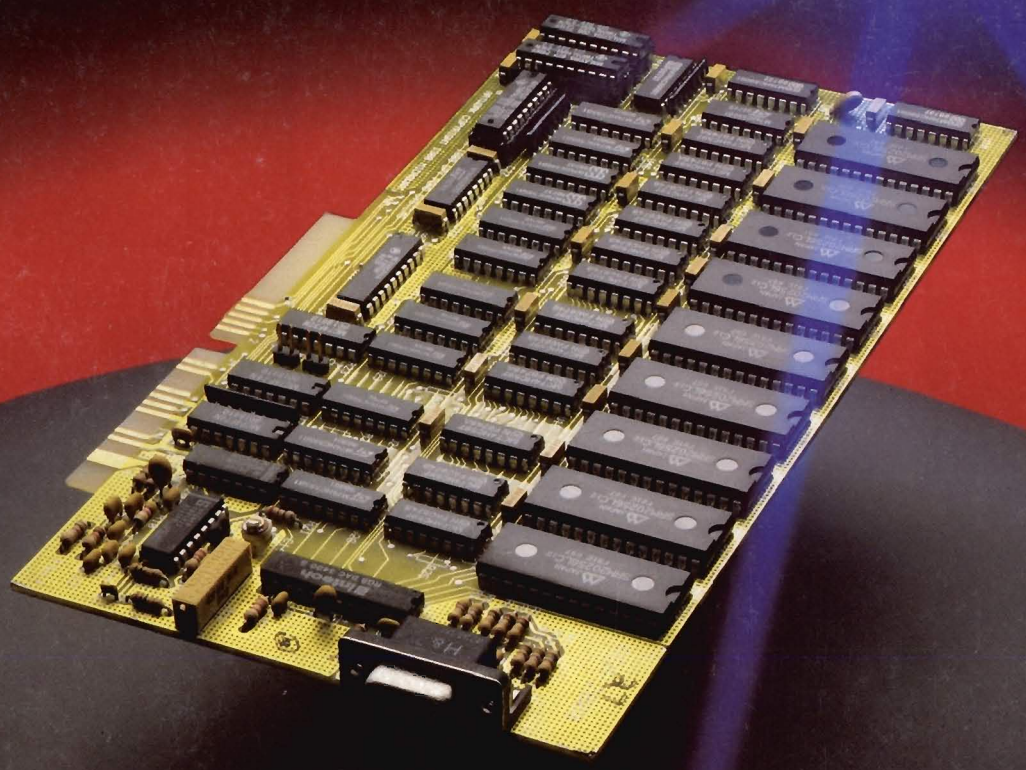
Expiry Date _____
Signature _____
Name _____
Address _____
_____
_____
Telephone _____

*ONTARIO RESIDENTS ADD 8%

T9046 *Please allow 28 days for delivery*

Photocopies of this coupon accepted

AGA 2000
The flickerFixer™



UNLOCK THE GRAPHICS POWER OF YOUR AMIGA 2000!

flickerFixer is an advanced graphics adaptor that eliminates your Amiga 2000's interlace flicker and visible scan lines. The result: superior quality color or monochrome graphics and text — for such demanding applications as CAD/CAM, Desktop Presentation, Graphic Design, Animation, 3D Modeling, Video, and Word Processing.

flickerFixer upgrades the Amiga 2000 with a flicker free 4096 color palette, has an overscan mode that features a screen size of 704 x 470 pixels and drives most of the popular PC Multiscan and VGA monitors, including the NEC Multisync and Mitsubishi XC1429C.

Accolades include: ***Best of 1988 Award,*** Commodore Magazine (12/88); ***#1 Reader's Choice Award***, AMIGAWORLD (12/88); ***"The display is fantastic . . . It is the best display we have ever seen on any computer system."*** Amiga GURU (5/88)

flickerFixer fits into the Amiga video slot, is fully compatible with all software, and does not modify the standard Amiga video signals. For more information or to order, call MicroWay Sales at (508) 746-7341 or your Amiga Dealer. Priced at $595, ***flickerFixer*** is made in the USA and is FCC Class B approved.

MicroWay . . . Respected throughout the industry for high quality engineering, service and technical support.

Micro Way

World Leader in PC Numerics

P.O. Box 79. Kingston, MA 02364 USA (508) 746-7341
32 High St., Kingston-Upon-Thames, UK, 01-541-5466
USA FAX 617-934-2414 Australia 02-439-8400

flickerFixer and MicroWay are trademarks of MicroWay, Inc. Amiga is a registered trademark of Commodore. Multisync is a registered trademark of NEC.